Performance Optimizations and Operator Semantics for Streaming Data Flow Programs

Dissertation
zur Erlangung des akademischen Grades
Dr. rer. nat.
im Fach Informatik

eingereicht an der

Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von

Dipl.-Inf. Matthias J. Sax

Präsidentin der Humboldt-Universität zu Berlin
Prof. Dr.-Ing. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät
Prof. Dr. Elmar Kulke

Gutachter
1. Prof. Johann-Christoph Freytag, Ph. D.
2. Prof. Dr. Odej Kao
3. Prof. Dr. Daniela Nicklas

Tag der Verteidigung: 28. Februar 2020

Abstract

Internet native companies are able to collect more data and require insights from it faster than ever before. This trend to online processing of giant data sets has not stopped at Internet giants, but nowadays affects data driven research and almost all businesses from finance and retail to classic manufacturers. Relational database management systems do not meet the requirements for processing the often unstructured data sets with reasonable performance. The database research community started to address these trends in the early 2000s. Two new research directions have attracted major interest since: large-scale non-relational data processing as well as low-latency data stream processing.

Large-scale non-relational data processing was pioneered by Google with their Google File System (GFS) and the MapReduce processing framework, and is commonly known as "Big Data" processing. While "Big Data" is characterized by the 4 Vs *volume*, *variety*, *velocity*, and *veracity*, in the beginning industry mainly focused on the challenge to handle large data sets. In parallel, low latency data stream processing was mainly driven by the research community developing prototype systems such as Aurora/Borealis, STREAM, and TelegraphCQ. Stream processing faced fundamental questions about semantics, incomplete data, and reasoning about time. The first generation of stream processing systems was not able to process high volume data streams, which made data stream processing a niche in its early years of development.

The MapReduce paradigm inspired a second generation of stream processing systems. The second generation embraces a distributed architecture, scalability, and exploits data parallelism. While these systems have gained more and more attention in the industry, there are still major challenges to operate them at large scale. Provisioning and performance tuning of queries needs to be carried out by experts, and is a manual, time consuming, and error prone process. Furthermore, there is still no agreement in research or in the industry for the semantics of continuous data stream processing, i. e., its data or operational model. Different systems offer different semantics and often lack deterministic query execution.

The goal of this thesis is two-fold. First, to investigate runtime characteristics of large scale data-parallel distributed streaming systems independent of their actual query semantics. And second, to propose the *Dual Streaming Model* to express semantics of continuous queries over data streams and tables.

Our goal is to improve the understanding of system and query runtime behavior with the aim to provision queries automatically. We introduce a cost model for streaming data flow programs taking into account the two techniques of record batching and data parallelization. Additionally, we introduce optimization algorithms that leverage our model for cost-based query provisioning.

The proposed *Dual Streaming Model* expresses the result of a streaming operator as a stream of successive updates to a result table, inducing a duality between streams and tables. A key challenge in distributed data stream processing is the inconsistency of the logical and the physical order of records within a data stream. Existing systems either ignore these inconsistencies or handle them by means of data buffering and reordering techniques, thereby introducing non-determinism or compromising processing latency. In our model, inconsistencies of logical and physical order are handled within the model itself, which allows for deterministic semantics as well as low latency query execution.

Zusammenfassung

Internetunternehmen sammeln mehr Daten als je zuvor und müssen auf diese Informationen zeitnah reagieren. Dieser Trend, riesige Datenmengen sofort nach der Erfassung zu verarbeiten, geht heute über Internetunternehmen hinaus und revolutioniert datengetriebene Forschung und fast sämtliche Wirtschaftszweige – vom Finanzsektor über das Gesundheitswesen und industrielle Produktion bis hin zu Medienunternehmen. Relationale Datenbankmanagementsysteme eignen sich nicht für die latenzfreie Verarbeitung dieser oft unstrukturierten Daten. Um diesen Anforderungen zu begegnen, haben sich in der Datenbankforschung seit dem Anfang der 2000er Jahre zwei neue Forschungsrichtungen etabliert: skalierbare Verarbeitung unstrukturierter Daten und latenzfreie Datenstromverarbeitung.

Skalierbare Verarbeitung unstrukturierter Daten, auch bekannt unter dem Begriff "Big Data"-Verarbeitung, wurde zunächst von Google mit dem "Google File System" (GFS) und "MapReduce" eingeführt. "Big Data" Datenverarbeitung, die mit den 4 Vs, "volume" (Volumen), "variety" (Vielfalt), "velocity" (Geschwindigkeit) und "veracity" (Richtigkeit) charakterisiert wird, hat in der Industrie schnell Einzug erhalten, wobei der Fokus auf der Verarbeitung riesiger Datenmengen lag. Zur selben Zeit wurden in der Forschung erste prototypische Systeme zur latenzfreien Datenstromverarbeitung entwickelt (z. B. Aurora/Borealis, STREAM und TelegraphCQ). Dabei wurden grundlegende Fragen zu Verarbeitungssemantiken, dem Umgang mit unvollständigen Daten und die Bedeutung der Zeitdimension addressiert. Die erste Generation von Datenstromverarbeitungssystemen war nicht in der Lage hochfrequente Datenströme zu verarbeiten und erhielten keinen breiten Einzug in die Industrie.

Basierend auf dem MapReduce Datenverarbeitungsparadigma wurde eine zweite Generation von Datenstromverarbeitunssystemen entwickelt. Die zweite Generation setzt auf eine verteilte Architektur, Skalierbarkeit und datenparallele Verarbeitung. Obwohl diese Systeme in der Industrie vermehrt zum Einsatz kommen, gibt es immer noch große Herausforderungen im praktischen Einsatz. Kapazitätsmanagement und Anfrageoptimierung werden manuell von Experten durchgeführt und sind fehleranfällig und zeitaufwending. Des Weiteren gibt es weder in der Forschung noch in der Indutrie einen standardisierten Ansatz für die Semantik von Datenstromverarbeitung, also kein einheitliches Daten- oder Operatormodell. Die Verarbeitungssemantik unterscheidet sich von System zu System, und die Anfrageergebnisse sind häufig nicht-deterministisch.

Diese Dissertation verfolgt zwei Hauptziele: Zuerst wird das Laufzeitverhalten von hochskalierbaren datenparallelen Datenstromverarbeitungssystemen untersucht. Im zweiten Hauptteil wird das *Dual Streaming Model* eingeführt, das eine Semantik zur gleichzeitigen Verarbeitung von Datenströmen und Tabellen beschreibt.

Das Ziel unserer Untersuchung ist ein besseres Verständnis über das Laufzeitverhalten dieser Systeme zu erhalten und dieses Wissen zu nutzen um Anfragen automatisch ausreichende Rechenkapazität zuzuweisen. Dazu wird ein Kostenmodell für Datenstromanfragen eingeführt, das Datengruppierung und Datenparallelität einbezieht. Aufbauend zu diesem Kostenmodell, stellt diese Dissertation verschiedene Optimierungsalgorithmen vor, um Datenstromanfragen automatisiert und kosteneffizient auszuführen.

Das vorgestellte Datenstromverarbeitungsmodell beschreibt das Ergebnis eines Operators als kontinuierlichen Strom von Veränderugen auf einer Ergebnistabelle, und induziert damit eine Dualität zwischen Datenströmen und Tabellen. Dabei besteht eine Hauptschwierigkeit im Umgang mit der Diskrepanz der physikalischen und logischen Ordnung von Datenelementen innerhalb eines

Datenstroms. Bestehende Systeme ignorieren diese Diskrepanz häufig oder lösen dieses Problem durch Datenpufferung und Umordnen von Datenelementen, was zu Nicht-Determinismus oder erhöhter Verarbeitungslatenz führt. Unser Modell behandelt die beschriebenen Diskrepanz als Teil des Modells und erreicht damit eine deterministische Semantik und eine minimale Verarbeitungslatenz.

Acknowledgments

First and foremost, I thank my advisor Prof. Johann-Christoph Freytag, Ph. D., for his many years of support. He became my primary mentor during my time as Diplom student and supported me ever since. For example, he helped me to land multiple internships in the US. After he sparked my interest in data management, data structures and algorithms, and conceptual thinking, he also encourage me to start a Ph. D. I always appreciated his high level of trust and freedom that allowed me to discover and pursue my personal research interests. Thank you Christoph!

I thank Prof. Dr. Odej Kao and Prof. Dr. Daniela Nicklas to take time out of their busy schedules to serve as my reviewers and I also thank all other members on the committee.

During my studies, I worked with many great people at the DBIS research group, the Stratosphere project, as well as the METRIK graduate school. Special thanks goes to Dr. Kostas Tzoumas, Dr. Malu Castellanos, and Prof. Dr. Matthias Weidlich for teaching me in the art of writing. I am grateful to Mathias Peters, Jörg Bachmann, Fabian Fier, Dr. Bruno Cadonna, and all other members of the DBIS research group for many fun hours discussing research and beyond. Thanks to all colleagues from the Stratosphere project, in particular Dr. Fabian Hüske, Dr. Stephan Ewen, Dr. Daniel Warneke, Dr. Astrid Rheinländer, and Dr. Arvid Heise. I learned a lot about computer science and programming from all of you. Also a big thank you to my colleagues at Confluent for their mental support and to my proof readers of this thesis: Arjun, Bruno, Ewen, Jesus, John, Konstantine, and Michael.

Finally, I thank my wife Marie for her unlimited support on my “crazy” ideas like starting a Ph. D. or to move to the US. I am looking forward to our future adventures!

Contents

List of Figures

List of Tables

Part I

Data Stream Processing

Chapter 1

Introduction

Contents

1.1 Motivation

In the last decade, data processing became increasingly important in research and industry because new technologies like modern mobile phones, sensors, and telecommunication systems (e. g., 5G standard) allow to generate, transmit, and store more and more data. For example, CERN stores about 350 PB in their data centers and their "Accelerator Logging Service" produces data streams of about 50 TB per week [CER17]. Furthermore, the advent of the Internet of Things (IoT) increases the need of real-time monitoring. The estimated number of connected IoT sensors by 2025 is projected to be around 80 billion [Cla15].

Additionally, for many business use cases it is paramount to analyze data in an online fashion to gain insight with low latency (often called "real-time processing"). Online trading is one example for which new information is highly valuable, while the value of information declines quickly over time. Another example is online fraud detection [BH02]: analyzing credit card transactions and deciding if a transaction is fraud *before* it is approved may reduce financial damage significantly. However, it requires that this decision can be made in the order of hundreds of milli-seconds. Less critical applications have an increasing demand to low latency data processing, too. For example, an airline offering a mileage-base status program wants to update a customer profile directly after a flight is completed, instead of hours or even days later.

In the past, there were three different types of systems that tackle different demands with regard to large-scale *or* low-latency data processing: (1) large-scale batch processing systems, (2) messaging, pub/sub systems, and (3) centralized stream processing system. However, none of those systems is able to address all of the aforementioned challenges. Inspired by the MapReduce paradigm, a new class of distributed, large-scale data stream processing systems emerged in recent

years with the goal to make large-scale data stream processing feasible. Those systems exploit data-parallelism and aim to process large volume data streams with low latency.

While first prototype systems were developed and used successfully by technology-savvy companies like Google or Yahoo!, data stream processing is not mainstream yet, because those systems are still very hard to deploy, maintain, and program. The most important question for application developers using those systems is, how much compute resources they need for a certain stream processing workload. This question is hard to answer even for software engineers at technology giants like Facebook: *"However, guessing the right amount of parallelism before deployment is a black art."* [CWI+16]. Another demand in industry is a standardized stream processing model similar to SQL for relational database system. While many models have been suggested in the past, none of them seems to fit all (or at least a majority) of use cases. The goals of this thesis are to (1) contribute to the understanding of the performance of distributed stream processing systems, (2) simplifying the deployment of stream processing programs, and (3) to unify existing stream processing models to enlarge the design space for stream processing applications.

1.2 Contributions

In this thesis, we first investigate the runtime behavior of distributed data-parallel stream processing systems, that execute continuous streaming programs and are expressed as data flows. To run a data flow program, users need to specify a configuration that is used to deploy the data flow program into the system. The program and system configuration need to be tuned manually to allow for an efficient and cost effective execution.

We introduce a rate-based cost model (Chapter 3) that describes CPU and network costs for the execution of data flow programs. Rate-based cost models are not new, however, they are mostly used to address classic query optimization problems. Similar to relational database optimization, continuous streaming queries can be rewritten to reduce execution costs. Operator reordering is often limited compared to relational queries, however, physical optimizations like choosing the best join algorithm apply in the same way.

Most existing cost models are based on centralized systems and do not apply to distributed systems. Furthermore, the optimization goal is usually to reduce execution cost at a logical and algorithmic level. In contrast, the goal of our cost model is to express the runtime costs at the system level. To this extend, we consider data parallel execution as well as record batching. Record batching is a system level optimization that reduces runtime overhead independent of query semantics and operator implementations (i. e., the algorithms used). Based on our cost model, we present various algorithms (Chapter 4), that are able to detects bottlenecks in a data flow program, predict data flow throughput, and compute an optimized configuration that avoids bottlenecks and minimizes latency.

In the second part of this thesis, we present the *Dual Streaming Model* (Chapter 5), that defines novel stream processing semantics with the goal to unify the benefits of existing approaches. The Dual Streaming Model unifies the concepts of data streams an relational tables in a holistic model, inducing a duality between

streams and tables. Relational tables are used to model the state of stream processing operators explicitly as first class citizen, in contrast to most existing models, that treat operator state as an internal implementation detail. Furthermore, the Dual Streaming Model makes explicit the inconsistency of the logical and physical order of records in a data stream, and handles this inconsistency *within* the model. Modeling operator state and record ordering explicitly, opens up the design space between processing latency, processing cost, and result correctness/completeness. Capturing those trade-offs within the model, (1) allows users to reason about query semantics, (2) emphasizes the temporal query semantics that are often neglected by other systems, and (3) allows users to pick different execution trade-offs for the same query without rewriting their program.

Parts of this thesis have been published in the following papers:

- **Matthias J. Sax**, Malu Castellanos, Qiming Chen, and Meichun Hsu. Performance Optimization for Distributed Intra-Node-Parallel Streaming Systems. In *29th International Conference on Data Engineering Workshops (ICDEW '13)*, pages 62–69, 2013.
- **Matthias J. Sax**, Malu Castellanos, Qiming Chen, and Meichun Hsu. Aeolus: An Optimizer for Distributed Intra-Node-Parallel Streaming Systems. (Demo) In *29th International Conference on Data Engineering (ICDE '13)*, pages 1280–1283, 2013.
- **Matthias J. Sax** and Malu Castellanos. Building a Transparent Batching Layer for Storm. *HPL Technical Report*. Hewlett-Packard Laboratories, HPL-2013-96, 2013.
- **Matthias J. Sax**, Guozhang Wang, Matthias Weidlich, Johann-Christoph Freytag. Streams and Tables: Two Sides of the Same Coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics (BIRTE '18)*, pages 1–10, 2018.

Other publications:

- Fabian Hueske, Mathias Peters, **Matthias J. Sax**, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the Black Boxes in Data Flow Optimization. In *Proceedings of the VLDB Endowment*, 5(11), pages 1256–1267, 2012.
- Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, **Matthias J. Sax**, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, 23(6), pages 939–964, 2014.
- **Matthias J. Sax**. Apache Kafka. Book chapter in *Encyclopedia of Big Data Technologies*, pages 1–8, 2019. Editors Sherif Sakr and Albert Zomaya. ISBN 978-3-319-63962-8.

1.3 Outline

The remainder of this thesis is structured as follows:

Part I – Data Stream Processing

Chapter 1: This chapter motivates the research questions addressed in this thesis. It lays out the contributions and related publications and outlines the thesis structure.

Chapter 2: We first contrast batch and stream processing to point out additional challenges in data stream processing. Additionally, we discuss principles of scalable, distributed data processing, followed by an introduction to the data and execution model of state-of-the-art distributed data-parallel stream processing systems. We formally describe our data model including streams and tables, and discuss order and time aspects. This chapter also contains related work to set the context for the core chapters of this thesis.

Part II – Cost-based Streaming Data Flow Optimization

Chapter 3: We introduce a cost model for data parallel streaming systems that considers CPU and network consumption. The cost model centers around the cost of a single operator in the data flow graph taking batching into account. We discuss how batching may increase operator throughput and describe inter operator dependencies. We discuss different batching approaches for data stream processing systems, including their advantages and disadvantages.

Chapter 4: This chapter builds on Chapter 3 and introduces several optimization algorithms exploiting our data flow cost model. We provide a holistic view of the overall cost with regard to the structure of the program, i. e., connections between operators. Given a configuration (parallelism and batch sizes), we show how our cost model can be used to detect bottlenecks in a data flow program. Additionally, we use our cost model to compute an optimal configuration for a target input data rate. We experimentally evaluate our cost model and algorithms considering the impact of batching on throughput and required parallelism.

Part III – Data Streaming Model

Chapter 5: The second part of this thesis proposes a novel stream processing model, combining streams and tables, and putting forward temporal processing semantics. The goal of our processing model is to decouple the processing latency from properties of the input streams, and to open the design space of stream processing applications by generalizing known concepts. Our model makes the trade-off between processing cost, processing latency, and result correctness/completeness explicit to the user and allows them to configure the system based on the application requirements.

Part IV – Discussion

Chapter 6: We conclude this thesis with a summary and final discussion of our contributions.

Chapter 2

Fundamentals: Distributed Parallel Data Flow Programs

Contents

Distributed data stream processing has gained a lot of interest in research and industry over the last decade due to the demand for low latency online processing of high volume data streams [CCA+10, NRNK10, GJPPMV10, LLP+12, TTS+14, ABC+15, CEF+17]. An often cited use case of distributed data stream processing is fraud detection and prevention [BH02]. For this use case, the earlier fraud is detected, the higher the business value. For example, low latency data stream processing allows for online monitoring of financial transactions like credit card usage. Monitoring transactions while they occur allows to not only detect, but even prevent fraud by declining a transaction immediately. Furthermore, new technologies like

the "Internet of Things" (IoT) [Cla15] generate large amounts of online data that needs to be processed with low latency.

The development of scalable stream processing system was inspired by Google's distributed file system GFS (Google File System) [GGL03] and Google's MapReduce framework [DG04, Dea06, DG08] that introduces a novel programming and execution model for scalable batch processing systems. MapReduce and similar systems [IBY+07, YDHP07, BEH+10, BCG+11, ZCD+12] are tailored to large scale batch processing. However, they are not well suited for low latency continuous stream processing leading to the development of scalable stream processing systems [NRNK10, GJPPMV10, LLP+12, TTS+14].

In this chapter, we first give a conceptual introduction to stream processing and contrast it to batch processing (Section 2.1). Afterwards (Section 2.2) we discuss the basic principles like parallelism, scaling, and system architectures that are relevant for scalable stream processing systems. In Section 2.3, we give an overview of state-of-the-art scalable stream processing systems and introduce basic terms and definitions that we use in Chapter 3 and Chapter 4 to describe our cost model and optimization algorithms. The first sections in this chapter discuss stream processing in general, focus on low level system properties, and treat operators as black boxes. In contrast, Section 2.4 introduces a semantic model for streams and tables that is the foundation of the stream processing operator semantics defined in Chapter 5. Finally, related work is discussed in Section 2.5.

Preliminary Definitions

We distinguish between the set of natural numbers without zero and the set of natural numbers including zero. We denote the former with $\mathbb{N}$ and the later with $\mathbb{N}_0$.

- $\mathbb{N} = \{1, 2, 3, \dots\}$
- $\mathbb{N}_0 = \{0, 1, 2, 3, \dots\}$

We also use the term *data items* to describe an atomic unit of data that is processed at once if the actual data representation is not relevant for the discussion. For example a data item could be a tuple/record, object, document, or some other unit.

2.1 From Batch Processing to Stream Processing

In this section, we contrast stream processing and batch processing in general [BBD+02, GO03a, ScZ05]. Both processing techniques have very different data models which impacts the properties of their available operators and runtime models. Additionally to stream and batch processing, we briefly describe *micro-batching* [ZDL+13, DZSS14]. We do not consider micro-batching in this thesis, but we contrast it to batching and stream processing to clearly distinguish between those techniques.

Batch Processing: In batch processing, data is modeled as a finite unordered collection of data items, e. g., as a set of tuples in the relational model. The finite input data is completely available when data processing starts. Thus, it is possible to access data items multiple times or to rearrange the data layout (e. g., sorting or creating indices). A batch processing program, like a relational query or a MapReduce job [DG04, Dea06, DG08], terminates when all data is processed and produces a finite result. Semantically, the input represents immutable facts from a specific point in time called a *snapshot*. Hence, it is not possible to modify the input data during processing.

Before we explain the differences between batch and stream processing, we introduce *continuous queries* in the next paragraph.

Continuous Queries: In batch processing, queries are actively issued by the user in an ad-hoc fashion, process an immutable snapshot of data, and finish processing after a finite amount of time. Carney et al. describe this approach as the *Human-Active, DBMS-Passive*[1] (HADP) model [CcC+02] and contrast it to the *DBMS-Active, Human-Passive* (DAHP) model that they suggest for monitoring applications. In the DAHP model, the query input is not immutable but may change over time and the system updates the result accordingly. Hence, a query is *deployed*[2] into a system and runs forever if not terminated explicitly by the user.

DAHP queries are also called *continuous queries* [TGNO92, BBD+02]. Terry et al. describe continuous queries as follows:

> "The results of a continuous query is the set of data that would be returned if the query were executed at every instant in time." [TGNO92] (c. f. [GNOT92])

Hence, a continuous query is not evaluated over a single snapshot of input data but over every snapshot. Continuous queries may be re-evaluated from scratch for each snapshot to replace the previous computed result with the new result. However, this is a compute intensive and inefficient approach. Depending on the query semantics, it may be possible to evaluate a continuous query incrementally and compute a delta between two consecutive snapshots that are used to update the query result [TGNO92, LPBZ96, LPT99].

Stream Processing: In stream processing, data is modeled as an unbounded ordered collection of data items, i. e., a potentially infinite sequence, called a data stream [BBD+02]. The data items in a data stream are immutable facts and only new data can be appended to the stream.[3] Hence, a data stream captures events over time in contrast to the snapshot model in batch processing. The potentially infinite input is not necessarily available when data processing starts and new input data may be added at any point in time. Because input data is infinite, it can be read only once using a linear scan. Parts of the input data may be buffered, but available space is limited to an arbitrarily large, but finite amount.

[1] *DBMS* stands for "Data Base Management System".
[2] We use the term *deployed* for DAHP queries to distinguish them for HADP queries.
[3] Some models also allow retraction and in-place updates [AAB+05].

Definition 1 (Stream Processing Program)**.** *A* stream processing program *is a special form of a continuous query (c. f. paragraph "Continuous Queries" above) that takes an infinite data stream as input and produces a potentially infinite result.*

All "updates" to the query input are appends to the data stream each triggering an update to the computed result. If all currently available input data is processed, a stream processing program waits until new data becomes available for processing. Because the input is potentially infinite, stream processing systems can only execute continuous queries that can be computed incrementally.

Micro-batching: Micro-batching [ZDL+13, DZSS14] is a stream processing approach based on batch processing that mimics stream processing. In micro-batching, the potentially infinite input stream is split into finite batches and a batch processing program is triggered for each input batch. Batches are kept as small as possible to achieve low processing latency. Systems like Spark Streaming [ZDL+13] define batch sizes based on system wall-clock time and may start to process a new micro-batch in one-second intervals.

Smaller batch sizes are difficult to achieve, especially in a distributed system like Spark, because smaller batches increase the processing overhead. Processing a micro-batch must be finished before the next micro-batch is ready for processing. However, triggering a batch processing job for each micro-batch includes a certain startup/deployment overhead [ADT+18]. Thus, if the batch size is too small, this deployment overhead dominates the execution time resulting in decreased system throughput. Since there is a minimum batch size that allows for efficient processing, there is also a minimum processing latency, due to the linear relationship between both. In practice, the deployment overhead forbids processing latencies below 500 ms [VPO+17]. Some use cases require lower processing latencies and hence, micro-batching can only be use for a fraction of streaming applications.

The micro-batching execution model is an infinite collect-deploy-process loop. First, the data for one micro-batch is collected, and second, a batch processing job is deployed to process this micro-batch of data. Because micro-batches are accumulated based on system wall-clock time, processing is inherently non-deterministic. Additionally, micro-batching provides different semantics compared to stream processing. For example, window-processing is based on full micro-batches and data from a single micro-batch cannot be divided into two different windows. We do not consider micro-batching in this thesis and point out that record batching in stream processing—as discussed in the next paragraph—is *not* related to micro-batching.

Record Batching in Stream Processing Systems: Record batching in stream processing systems is a buffering technique that allows for an efficient execution of streaming programs [CcC+02, LWK12]. In contrast to micro-batching or batch processing, record batching is not part of the processing model but an implementation detail and is a well established technique in many different systems. For example, Aurora [CcC+02, CcR+03] uses a record batching technique called *train scheduling* to "describe the batching of multiple tuples as input to a single box"[4]. The Nephele

[4] A "box" in Aurora represent an operator.

Table 2.1: Properties of Batch, Continuous, Stream, and Micro-batch Processing

	batch	continuous	streaming	micro-batch
input size	finite	finite+updates or infinite	infinite	infinite (split into finite batches)
output size	finite	finite+updates or infinite	infinite	infinite
evaluation	holistic	holistic or incremental	incremental	incremental
latency	high	very low to high	very low	low
query runtime	finite	infinite	infinite	infinite
ordered input	no	maybe	yes	yes
deterministic	yes	yes	yes (not always)	no

system [LWK12] uses buffering in the network layer to increase the system throughput as record batching technique. In this thesis, we use the term *batching* in this sense and use batching as optimization technique to increase system performance. We refer to Chapter 3, for a detailed discussion of our used batching techniques.

Table 2.1 summaries the discussed properties of batch processing, continuous queries, stream processing, and micro-batching. We have seen that batch and stream processing are built on different assumptions. Based on these assumptions we describe corresponding implications that are relevant for this thesis in the next section. Those implications build the foundation of our streaming data flow cost model that we introduce in Chapter 3.

2.1.1 Properties of Stream Processing Systems and Batch Processing Systems

The main difference between batch and stream processing is finite and infinite input data. This difference has implications on operator properties, memory requirements, as well as performance metrics. We discuss those implications in the following paragraphs.

Blocking vs. Non-Blocking Operators: A blocking operator is an operator that cannot produce any output data until it has processed all the input data [BBD+02, LWZ04]. Shanmugasundaram et al. [STD+00] relax the definition of blocking operators and allow blocking operators to output partial results early, i. e., before all input data is processed. However, Shanmugasundaram's definition still states that a blocking operator needs to process the complete input before it can emit the complete result. For example, an outer-join may emit the partial inner-join result early, i. e., before all input data is consumed. It still must process the whole input before it can output the records that did not join.

Both definitions imply that blocking operators cannot be used in stream processing because the input is potentially infinite, and thus, a blocking operator can

never generate the complete (or even any) output. It is important to note that some operators may have different implementations that may be blocking or non-blocking. Therefore, an operator is considered non-blocking if at least one non-blocking implementation exists. For example, an inner-join can be implemented as a sort-merge-join that is a blocking implementation. As an alternative, it can be implemented as a symmetric hash-join [AA91] that is non-blocking. Therefore, inner-join is considered a non-blocking operator.

Memory requirements: In batch processing the input is finite, and thus, operators naturally require finite memory only. However, for some batch processing operators, memory requirements grow with the input data set size. For example, an inner-equi-hash-join builds up a hash-table for one input with a hash-table size that is linear to the input data set. Thus, those operators cannot be applied to an infinite input data stream as this would result in unbounded memory usage. Nevertheless, some operators (like joins) are conceptually useful for data streams, too. Therefore, different techniques were suggested to provide "streaming versions" of those operators [BBD+02]. Those techniques bound the space requirements of batch processing operators on input data streams to make the operators applicable to data stream processing. The available techniques can be categorized into (1) result approximation techniques [DGGR02, JMR05] and (2) operator re-definitions to provide a streaming version of an operator. The most common operator re-definition is windowing [GO03b, LMT+05]. Windowing limits the "scope" of an operator to finite subsets of the infinite input data stream. Using our example of an inner-equi-join, a windowed inner-equi-join is basically a band-join with an additional join condition that is implicitly defined by the join window.

Most stream processing systems use windowing to limit memory consumption because it is easy for users to reason about the well-defined expected result. It is important to note that even if windowing limits the result, this limitation is part of the operator definition, and thus, there is a notion of result completeness. Result approximation on the other hand is less common in practice. Partly because the result may be non-deterministic, hard to predict, or the result completeness cannot be guaranteed.

Execution Time, Throughput, and Latency: In batch processing, programs terminate after all input data is processed. Performance of a batch processing system is usually measured as *execution time*, i. e., the time it takes for the program to finish processing. In stream processing, programs run forever, and thus, execution time is naturally infinite and hence not useful to measure the performance of a stream processing system. Instead, *throughput* and *latency* are used to compare performance. Throughput is the amount of input data (i. e., number of data items) per time unit a system can process, and latency is the amount of time it takes until new input data is reflected in the output (i. e., the time it takes to process a single data item) [CcC+02]. While throughput is usually measured as an average number, latency is measured as mean, percentile, or even maximum processing time per data item.

The discussed implications of operator properties, memory requirements, and performance metrics impact the cost model of this thesis. In particular the notion of throughput and latency. We describe this impact in the next section.

2.1.2 Cost Model Considerations

In this thesis, we introduce a cost model for the execution cost of continuous queries over data streams (Chapter 3). Our cost model is based on the properties of stream processing as introduced in Section 2.1.1. In the following, we discuss the impact of those properties on our cost model and contrast it to cost models in batch processing.

In batch processing systems (like relational database systems), cost models estimate the execution cost of a query as the overall cost to compute the query result. The goal of those cost models is to *rank* different execution plans that are generated via logical and/or physical optimization techniques. However, those models are not applicable to stream processing systems, because queries run forever and costs would be estimated as infinite. Furthermore, compared to batch processing, logical and physical optimization is limited in data stream processing due to the continuous execution property, different operator semantics, and infinite input data.

In this thesis, the optimization goal is to find a configuration[5] for a given streaming program, that *provisions* the corresponding query execution plan based on the data rates of the input data streams. We model costs as "time units" to process or transfer records and introduce the concept of *capacity* as "amount of work by time unit" [CcC+02] (c. f. Chapter 3). Hence, our cost model is throughput-based [VN02] and there is no notion of overall query cost. Furthermore, we apply our cost model to distributed and scalable data stream processing systems. We cover general principles of distributed data processing in the next section and introduce scalable stream processing systems in Section 2.3.

2.2 Principles of Distributed Data Processing

Scalable data processing is a well understood topic and state-of-the-art scalable stream processing systems leverage the same concepts as scalable batch processing systems or parallel relational database systems. Understanding concepts like different levels of parallelism (Section 2.2.1), scaling techniques (Section 2.2.3), and system architectures (Section 2.2.4) is a requirement to design a processing cost model for continuous streaming queries.

2.2.1 Parallelism

In order to scale data processing systems (Section 2.2.3) multiple different types of parallelism can be exploited: pipeline parallelism, operator parallelism, and data parallelism [DG92]. Those three types of parallelism can be categorized as *inter* operator parallelism and *intra* operator parallelism [OV99]. Inter operator parallelism occurs, if two different operators can be executed in parallel. On the other hand, intra operator parallelism implies that parallelism can be exploited within a

[5] We formally define a *configuration* in Chapter 4.

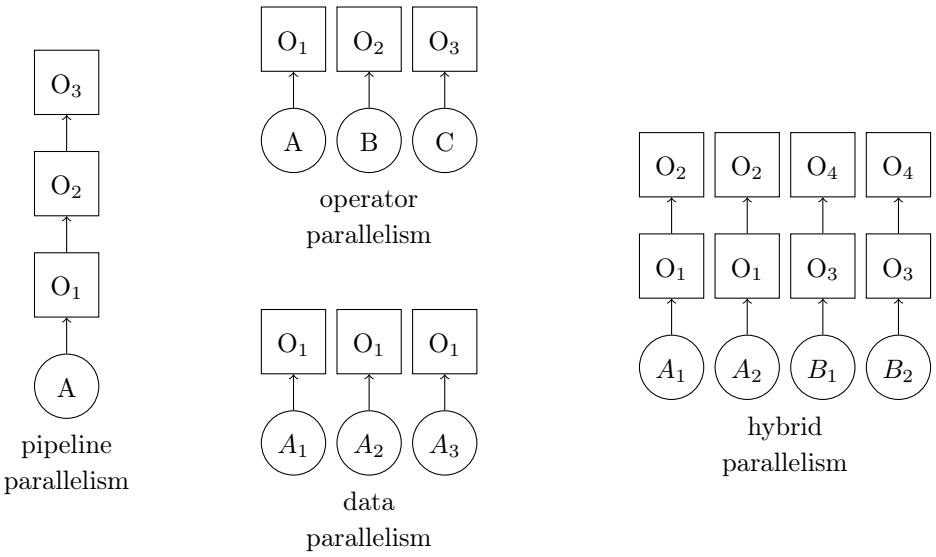

Figure 2.1: Types of parallelism.

single operator. The three types of parallelism are explained and categorized in detail below and illustrated in Figure 2.1. In Figure 2.1, data sources are depicted as circles, and different letters indicate different logical data sources, e. g., data sources A and B. Data sources with the same letter but different indices depict one logical data source that consists of multiple physical sources, e. g., data source A_1 and A_2 (c. f. paragraph *data parallelism* below). Operators are represented by squares and each operator may be executed in parallel to all other operators. Different indices indicate different operators, e. g., O_1 and O_2. If an index is used more than once, it implies that the same operator is executed multiple times with different input data.

Pipeline Parallelism Pipeline parallelism is possible for non-blocking operators only. It can be exploited if an upstream operator's (partial) output can be processed by a downstream consumer in parallel to the upstream operator. As pipeline parallelism involves two different operators, it belongs to the category of inter operator parallelism. For example, if there are two consecutive filter operators, the second filter operator can be executed, even if the first filter operator did not yet process its complete input. In batch processing, not all operators allow for pipeline parallelism as some operators are blocking. For example, a sort and a downstream aggregation operator cannot exploit pipeline parallelism as the blocking sort operator cannot emit any output data before it has consumed its entire input. In stream processing all operators must be non-blocking and therefore pipeline parallelism is a native property of stream processing programs.

Operator Parallelism Operator parallelism occurs, if two operators are *independent* of each other, i. e., each operator can process its input data independent of any other operator. Thus, operator parallelism is categorized as inter operator parallelism. In batch processing, operator parallelism allows for a higher degree of freedom in operator execution order as parallel operators may be executed one after another (in any order), concurrently, or in parallel. Thus, operator parallelism

can be exploited to increase parallelism by executing independent operators in their own threads. In stream processing, operator parallelism can be exploited, too. For example, if a program receives two input streams both streams can be processed independently as long as they are not merged or joined with each other.

Data Parallelism Data parallelism means that the input data can be partitioned (c. f. Section 2.2.2 below) and each partition of the input data can be processed independently of all other partitions. The system can start multiple instances of the same operator and use each operator instance to process one partition. Hence, in contrast to operator and pipeline parallelism, data parallelism allows for intra operator parallelism. Data parallelism is the fundamental concept in scalable data processing and is used in parallel relational databases [DG92] as well as in MapReduce [DG04, Dea06, DG08] and related batch [BEH+10, ZCD+12] and stream processing systems [NRNK10, GJPPMV10, LLP+12, TTS+14].

Programs may also combine different categories of parallelism. For this case, we use the term *hybrid parallelism*. To exploit parallelism, each operator is executed in its own thread. In this model, pipeline parallel operators are connected via FIFO (first-in-first-out) queues [MF02, BBD+02, CcC+02, CcR+03, LLP+12] and the upstream operators write their output into the FIFO queues while the downstream operators reads their input from the FIFO queues. We discuss pipeline parallelism via FIFO queues in Chapter 3 in more detail.

2.2.2 Data Partitioning

In Section 2.2.1, we explained how data parallelism can be used to increase parallelism in data processing systems. However, we did not discuss how input data is distributed into partitions. Let n be the number of partitions and $p_0, \ldots, p_{n-1}$ denote the corresponding partitions. In the following, we explain data partitioning patterns for assigning records to partitions [DG92].

Random Data Partitioning Random partitioning implies that each record is stored in a randomly selected partition. Let rand(R) be a function that returns a random number r between $0 \leq r \leq R - 1$. For each record, the function is used to compute a partition p_i with $i = \text{rand}(n)$. Random partitioning can also be implemented via a round-robin algorithm instead of using rand(R).

Random data partitioning has the advantage that each partition contains about the same number of records, i. e., it achieves good load balancing. However, some operators may require data co-partitioning based on some criteria, and thus, random partitioning can only be used for a subset of operators.

Hash-based Data Partitioning For many operators it is required that a single operator instance processes a certain subset of the data. For example, if the operator computes an aggregation based on some grouping criteria (similar to a `group-by-aggregation` clause in SQL), it is required to store all records of the same group in the same partition. It is important to note that for this case, multiple groups are combined into one data partition.

Let $h(k)$ be a hash-function and $r.o$ be the grouping attribute of records r that defined the co-partitioning requirement. Using hash-based partitioning, each record is assigned to partition p_i with $i = h(r.o)\%n$. While hash-partition allows to co-locate data, it cannot guarantee good load balancing.

Range-based Data Partitioning Range-based partitioning is similar to hash-based partitioning and is used if certain subsets of the input data must be processed together. The difference to hashing is that instead of using a hash-function, the key-space[6] is divided into n ranges, and each range is associated with one partition. For each input record r, the grouping attribute's range is computed and the record is stored in the corresponding partition. If the key distribution of the input data is known, range-based partitioning may achieve better load balancing than hash-partitioning if the ranges are chosen accordingly.

Broadcast Data Distribution Broadcasting is the opposite to data partitioning. If a broadcast data distribution is used, data is replicated to all partitions. Broadcasting is a very expensive operation and usually only used for small data sets. It is only useful if a single operator processes multiple different data sets and at least one data set is not replicated but partitioned. It is also possible to combine broadcasting and partitioning into hybrid distribution strategies [SY93].

Which data partitioning strategy is used depends on the semantics of the operators that process the data. For more complex queries, it is often required to repartition data between consecutive processing steps. We discuss how data distribution strategies are applied in distributed stream processing systems in Section 2.3.

2.2.3 Scaling

The amount of work a computer system can handle per time unit depends on the system's hardware and software. A scalable computer system is a system that can be enlarged to accommodate a growing amount of work per time unit. Scaling is used if more data needs to be processed, if computation time should be reduced, or both.

Scaling a system always implies to add additional and/or more powerful hardware resources to the system. For example, a server may be replaced with a newer model that has a more powerful CPU (i. e., higher clock speed). However, adding more hardware resources may not be sufficient to scale a system, because the software must be scalable, too, i. e., it must be able to utilize all hardware resources. For example, if more CPU cores are added, those cores can only be utilized if enough threads are executed. A system running a single threaded program is not scalable by this means.

In the following, we distinguish between *vertical* and *horizontal* scaling:

Vertical Scaling Scaling a system vertically implies that more hardware resources or more powerful hardware are added to the system. Vertical scaling applies to a

[6] We use the term *key* to refer to the grouping criteria. Thus, a key is not a primary/unique key for the input data in this case.

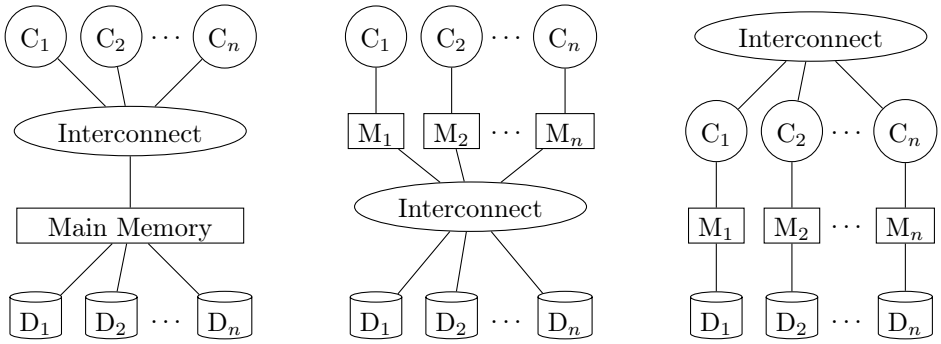

Figure 2.2: System architectures following DeWitt and Gray [DG92].

single server, i. e., it means a more powerful server is used. For example, one may add more powerful CPU, increase the main memory, or increase disk space to a system to scale it. Vertical scaling is called *scaling up* if resources are increased and called *scaling down* if resources are decreased. The advantage of vertical scaling is that it can improve system performance without rewriting the software. For example, if more main memory is added, all data might fit into memory and swapping data to disk might not be required anymore. However, vertical scaling is limited by the maximum capacity of a single server.

Horizontal Scaling Another scaling technique is horizontal scaling. In contrast to vertical scaling, horizontal scaling does not imply that a more powerful server is used, but that new servers are added to a compute cluster that forms a single system. Thus, horizontal scaling applies to distributed systems, while vertical scaling applies to single server systems.[7] Horizontal scaling is called *scaling out* if resources are added and called *scaling in* if resources are removed. The main benefit of horizontal scaling is that it is unlimited from a hardware perspective. It is always possible to add another server to the system. In practice, horizontal scaling is limited though, as cross system communication introduces some overhead that grows with the size of the overall system. Furthermore, horizontal scaling requires a different software architecture to be able to exploit the available hardware resources efficiently.

Horizontal scaling is not applicable to all system but depends on the system's architecture, as shown in the next section. It is noteworthy that there is a relationship between horizontal scaling and data parallelism (Section 2.2.1) because data parallelism allows for horizontal scaling. This relationship is exploited by scalable stream processing systems as discussed in detail in Section 2.3.2.

2.2.4 System Architecture

Modern batch and stream processing systems exploit horizontal scaling and data parallelism (c. f. Section 2.2.1 and Section 2.2.3) to address the needs of "Big Data"

[7] A compute cluster may also be scaled vertically, by replacing existing servers within the cluster with more powerful ones.

processing as laid out in the introduction of this thesis. To achieve both, they implement a so-called *shared nothing* system architecture as introduced by DeWitt and Gray [DG92]. Besides a shared nothing architecture, DeWitt and Gray also distinguish shared memory and shared disk systems (Figure 2.2). In this section, we discuss the advantages and disadvantages of those three architectures and explain why the shared nothing architecture is the dominant pattern in scalable data processing systems.

Shared Memory Single server systems are so-called shared memory systems. Those systems can have one or multiple CPUs with one or multiple cores each. All available cores share the same main memory address space. Because main memory is shared, data exchange between different threads is cheap, however, it requires synchronized data access between threads. The disadvantage is that shared memory systems can only be scaled up vertically; horizontal scaling is not applicable.

Shared Disk In shared disk systems, there are multiple servers with their own CPU and main memory resources. Thus, there is no global main memory address space, but each server has its own address space. All servers share a global pool of disks called a storage area network (SAN). This allows for data exchange between servers via writing/reading to/from files. Those reads and writes must be synchronized, similar to in-memory data access as in a shared memory architecture. Shared disk system are loosely coupled compared to shared memory systems and horizontal scaling is possible to some extent because new servers can be added easily. However, the required synchronization for disk-based data access limits their horizontal scalability.

Shared Nothing In this architecture, each server has its own main memory and disks and all servers are connected via network to each other. Local disks can be accessed more efficiently compared to a shared disk architecture with SAN. Thus, shared nothing clusters are easy to scale horizontally as no cross-server synchronization is required: each server has its own local main memory and disk. The disadvantage is the potentially more expensive data exchange via network. Furthermore, if multiple servers need to access the same data, each server must have its own copy (c. f. paragraph *Broadcast Data Distribution* in Section 2.2.2) of the data, resulting in increased disk usage. In practice, shared nothing architectures are used for data parallel processing, and thus, sharing data is a limited concern for this case.

Modern scalable stream processing systems implement a shared nothing architecture [NRNK10, GJPPMV10, LLP+12, LWK12, TTS+14] that is well suited for cluster and cloud based deployments. Especially in the cloud, horizontal scaling is simplified as new virtual machines can be added to the cluster easily. As mentioned in Section 2.2.3, scalable stream processing systems try to exploit data parallelism to allow for horizontal scaling. Implementing a shared nothing architecture aligns with this design.

2.3 Scalable Stream Processing Systems

In this section, we discuss state-of-the-art stream processing systems [NRNK10, GJPPMV10, LLP+12, LWK12, TTS+14] that we study and advance in this thesis. In Section 2.3.1, we introduce the programming model that is used by most of those system. Additionally, we discuss the execution model (Section 2.3.2) based on the scaling and system architecture principles from Section 2.2.3 and Section 2.2.4. We build on both models (programming and execution model), in the remainder of this thesis.

2.3.1 Data and Programming Model

In stream processing, data is modeled as a *data stream* which is an infinite sequence of data items. There are a large variety of data formats for those data items. Some models define them as tuples (t, e) [KS09], where t is a timestamp that defines the logical order of a record within the stream and e is the actual payload to be processed. Others use a key-value based model [LLP+12, ABB+13, NPP+17]. Most systems define records similar to relational n-ary tuples with a schema and atomic or complex data types [ACc+03b, ABW06, Gul12, MMI+13, TTS+14]. We refer to data items as *records* ignoring the actual data format in Chapter 3 and Chapter 4. Furthermore, we do not reason about operator semantics but model operators as black boxes in both chapters. Operators are called for each input record once, and may emit an arbitrary number of output records in each call. In Section 2.4 we cover existing data and programming models in more detail, and we introduce a novel processing model with well defined operator semantics in Chapter 5.

Data flow programs are a common abstraction in distributed scalable data processing systems and used in batch [IBY+07, BEH+10, BCG+11, ZCD+12] as well as in stream processing systems [AAB+06, NRNK10, LLP+12, Gul12, LWK12, MMI+13, TTS+14]. Formally and independent of a batch or stream processing context, we define a data flow program as follows:

Definition 2 (Data Flow Program)**.** *A* data flow program $D = (V, E)$ *is a connected directed acyclic graph (DAG) consisting of a set of vertices V (called nodes) that model operators and a set of directed edges $E \subseteq V \times V$ that model data flow between operators.*

Edges in the data flow go from one (upstream) producer *node to one or multiple (downstream)* consumer *nodes.* Source *nodes do not have any upstream producers, i. e., no incoming edges.* Sink *nodes do not have any downstream consumers, i. e., no outgoing edges. All nodes that are neither sources nor sinks are called* processing nodes*.*

It is also possible to allow for cycles in the data flow program [LLP+12]. However, those may result in "infinite processing loops" if not programmed carefully. To avoid this problem, some systems limit the usage of cycles to a restricted set of system controlled loops [MMI+13, CEF+17].

By default, each consumer receives a full copy of the producer's output data (called *broadcasting*). Thus, during execution it might be required to duplicate the producer's output data (i. e., one copy for each consumer). Most systems also allow to split a data stream into multiple output streams and send each output stream

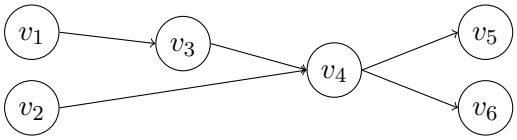

Figure 2.3: Example data flow program with six nodes.

to a different consumer. For this case, data duplication is not required. Hybrid models that allow to send a record into an arbitrary number of output streams are also possible. There are two special types of nodes: *sources* and *sinks*. They are responsible to connect to other systems. While sources ingest input data for the computation, sinks publish the computed result.

Example 1. *Figure 2.3 shows a data flow program with six operators and is defined as* $D = (V, E)$ *with*

- $V = \{v_i | 1 \leq i \leq 6\}$ *and*
- $E = \{(v_1, v_3), (v_2, v_4), (v_3, v_4), (v_4, v_5), (v_4, v_6)\}$

It has two sources v_1 *and* v_2 *and two sinks* v_5 *and* v_6*. Processing node* v_3 *has one producer (source* v_1*) and one consumer (*v_4*). The second processing node* v_4 *has two upstream producers (*v_2 *and* v_3*) and two downstream consumers (sinks* v_5 *and* v_6*).*

Based on Definition 1 and Definition 2, we define a *streaming data flow program* as follows:

Definition 3 (Streaming Data Flow Program)**.** *A* streaming data flow program $D = (V, E)$ *is a data flow program that models a stream processing program.*

All operators of a streaming data flow program must be non-blocking operators, to allow for a continuous and incremental computation over potentially infinite input data streams. Hence, we model operators as a function $f : r \mapsto \{r_1, \ldots, r_n\}$ with $n \in \mathbb{N}_0$ that takes a single input record r and emits zero or more output records $r_1, \ldots, r_n$. For example, an operator may take a record containing a sentence as input and emit an output record for each word in the sentence.

In the remainder of this thesis, we always assume streaming data flow programs even if we often refer to them as data flow programs for simplicity. Similarly, we use the term "continuous query" to refer to continuous streaming queries.

2.3.2 Program Execution

To execute a data flow program, distributed stream processing system exploit multiple different types of parallelism: pipeline parallelism, operator parallelism, and data parallelism. While pipeline and operator parallelism are expressed in a streaming data flow program (Definition 3) already, data parallelism is not represented there. As discussed in Section 2.2.1, data parallelism can be exploited by partitioning the input data (Section 2.2.2) and running multiple operator instances in parallel. In data stream processing, the input stream is split (i. e., partitioned) into multiple substreams and one operator instance may be executed per substream. Executing

multiple operator instances within a data flow program in parallel requires to transform the program into an *execution graph* that is a parallelized version of the data flow program. For this parallelization, each operator gets a *degree of parallelism* (*dop*) assigned that defines how many instances of this operator should run in parallel. We call each operator instance a *task*. Each edge (v_i,v_j) in D from a producer v_i to a consumer v_j is translated into edges from all tasks corresponding to v_i to all tasks corresponding to v_j in the execution graph. Formally, we define an execution graph as follows:

Definition 4 (Execution Graph)**.** *Let $D = (V, E)$ be a data flow program as defined in Definition 3 and*

$$dop : V \to \mathbb{N}$$

be a function assigning a degree of parallelism *to each node of D. The corresponding* execution graph *$EG = (T, F)$ of D and dop is a directed acyclic graph (DAG) with a set of vertices T (called* tasks*) and a set of directed edges $F \subseteq T \times T$ (called* connections*) with*

- $T = \{t_j^i | v_i \in V \wedge j \in [1; dop(v_i)]\}$
- $F = \{(t_k^i, t_l^j) | (v_i, v_j) \in E \wedge k \in [1; dop(v_i)] \wedge l \in [1; dop(v_j)]\}$

The semantics of an execution graph are similar to the original data flow graph. Nodes model operators and edges model data flow connections. One important difference between a data flow graph and an execution graph is, that in general, for each producer-consumer pair exactly one consumer task receives an output record that can be emitted by any producer task. This pattern is derived from the underlying data parallelism concept. All tasks of a consumer perform the same operation and data parallelism dictates that each record is processed once—i. e., by exactly one task. Depending on the consumer operator, different data partitioning/distribution patterns are possible (c. f. Section 2.2.1). We assume that data must be redistributed between the tasks of each consumer-producer pair in the original data flow program. Otherwise, the local-forward pattern [GJPPMV10] may be used. The local forward pattern requires that the number of producer tasks is equal to the number of consumer tasks. Furthermore, not all producer-consumer-task-pairs are connected to each other, but each producer task is connected to exactly one consumer task. The local forward pattern has the advantage that producer and consumer tasks can be co-located on the same physical machine, avoiding an expensive network communication channel between both. The disadvantage is that local forward pattern might suffer from imbalanced load. It is important to note that the local forward pattern allows for operator fusion[8] [CcC+02, CcR+03, CRP+10, HSS+14, LWK12, SGH15] and therefore we do not consider it, but assume that operator fusion was applied before the data flow graph was translated into the execution graph.

The most common data distribution patterns in stream processing are random, hash, or range partitioning. For some cases, a broadcast distribution is required. However, broadcasting does not align with the idea of data parallel processing. In

[8]Operator fusion is also called *combining boxes/operator batching* or *task chaining* in the literature. We discuss operator fusion in more detail in Section 3.2.1.

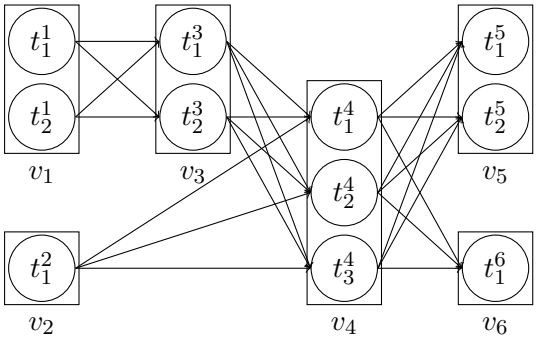

Figure 2.4: Execution graph from Example 2.

practice, broadcasting is used very rarely in data stream processing and if used, it is usually only applied to data streams with small data rates.

In the remainder of this thesis, we mostly ignore the concrete distribution pattern but build upon the observation that each record is processed by exactly one task for each node, i. e., we ignore the broadcast pattern. Furthermore, we assume a load-balanced data distribution over all tasks of a node. Example 2 shows the translation of a data flow program into an execution graph as defined in Definition 4.

Example 2. *Given the data flow program $D = (V, E)$ from Example 1 and dop : $V \to \mathbb{N}$ as*

- $dop(v_1) = 2, \quad dop(v_2) = 1, \quad dop(v_3) = 2,$
  $dop(v_4) = 3, \quad dop(v_5) = 2, \quad dop(v_6) = 1$

the execution graph from D with respect to dop is defined as $EG = (T, F)$ with

- $T = \{t_1^1, t_2^1, \quad t_1^2, \quad t_1^3, t_2^3, \quad t_1^4, t_2^4, t_3^4, \quad t_1^5, t_2^5, \quad t_1^6\}$ *and*
- $F = \{(t_1^1, t_1^3), (t_1^1, t_2^3),$
  $(t_2^1, t_1^3), (t_2^1, t_2^3),$

  $(t_1^2, t_1^4), (t_1^2, t_2^4), (t_1^2, t_3^4),$

  $(t_1^3, t_1^4), (t_1^3, t_2^4), (t_1^3, t_3^4),$
  $(t_2^3, t_1^4), (t_2^3, t_2^4), (t_2^3, t_3^4),$

  $(t_1^4, t_1^5), (t_1^4, t_2^5), (t_1^4, t_1^6),$
  $(t_2^4, t_1^5), (t_2^4, t_2^5), (t_2^4, t_1^6),$
  $(t_3^4, t_1^5), (t_3^4, t_2^5), (t_3^4, t_1^6)\}$

Figure 2.4 shows the execution graph from Example 2. Each node in the data flow graph may have a different *dop*. For example, node v_4 has a *dop* of three, hence, it is executed with three tasks, t_1^4, t_2^4, and t_3^4. All three tasks are connected to t_1^5 and t_2^5 of v_5 and to t_1^6 of v_6, since v_5 and v_6 are consumers of v_4. Furthermore, all three tasks t_x^4 are connected to all tasks of nodes v_2 and v_3, because v_4 is a consumer of v_2 and v_3. As explained above, there are two different data distribution patterns that overlay with each other: the broadcasting or splitting *data flow* pattern, as well as the data parallel *execution graph* pattern. In our example, most nodes have only

one consumer, and thus, their tasks are only subject to the data parallel distribution pattern. However, the tasks t_x^4 are subject to both patterns, because node v_4 has two downstream consumers. Thus, if any task t_x^4 emits a record, it will send it to one task t_y^5 and to one task t_z^6, assuming that the logical distribution pattern is a broadcast. The broadcast pattern applies to the data flow (logical) representation, meaning, each output record is sent to all consumer *nodes*. Broadcast does not imply sending a record to all *tasks*. For the execution graph data-parallelism applies and exactly one task per node receives a record.

In Chapter 3 and Chapter 4, we model the cost of streaming data flow programs and their execution graphs. One goal is to find an optimal parallelization for a data flow program, i. e., translation into an execution graph, given the data rates of the input data streams. In the next section, we introduce fundamental concepts that we use in Chapter 5 to define our stream processing model.

2.4 Data Streaming Model

The success of relational database technology is built on the relational data model [Cod70], which provides a strong mathematical foundation and well defined deterministic operator semantics. This strong foundation allowed SQL to become a standard query language widely adopted in research and industry. However, in data stream processing, there is no unique or standardized model of the semantics of the computation. While formal models exist [SLR95, BBD+02, ACc+03b, ABW06, BGAH07, JMS+08, KS09, LFQ+16], there is no standardized terminology and therefore we introduce the formal notation of our streaming data model in the following sections. Based on those concepts, we formally define our stream processing model in Chapter 5.

First, we introduce a formal model for streams and tables inspired by existing literature [Cod70, SLR95, ABW03, SW04, JMS+08]. Since there is no unique naming convention in the literature, we introduce the terms we are using and point out synonyms as used by others (c. f. Table 2.2). We refer to data items as *records* in our model. Records are ordered by their unique *offset* (often called position or sequence number) in a stream [SLR95, LFQ+16]. Additionally, each record has a scalar timestamp from the discrete time domain $\mathcal{T}$ [SW04, BGAH07] assigned. Some other models use multiple timestamps [BGAH07, CGB+14] or time intervals [KS09, LFQ+16]. We use scalar timestamps as they are sufficient for the model we propose. Without loss of generality, we assume $\mathcal{T} = \mathbb{N}_0$ [ABW03].

There are two common time concepts: *processing-time* and *event-time* [SW04, BGAH07]. Processing-time (also called system-time, server wall-clock time, or physical-time) is the time when a record is processed. It is used to define semantics for time-based operators if a records does not have an embedded timestamp in its payload. Event-time (also called application-time, occurrence-time, or external-time) is the time when a record is *created*. If event-time shall be used to define operator semantics, the event-time timestamp must be added to the record to expose it to the processing system. We assume that a *producer* (i. e., an external data source application that pushed record into the processing system) adds the event-timestamp before appending the record to the data stream. Not all producers

Table 2.2: Used Terminology and Synonyms as used in Related Work

Used Term	Synonym
Record	Event, Message, Tuple
Offset	Sequence Number, Position
Processing-Time	System-Time, Server Wall-clock Time, Physical-Time
Event-Time	Application-Time, Occurrence-Time, External-Time
Ingestion-Time	Arrival-Time, Internal-Time
Producer	Data Source

may have a clock, and thus, may not be able to add a timestamp to a record. For these cases, event-time can be approximated by the processing system: when data is ingested into the system, the current system time (i. e., *ingestion-time* timestamp; also called arrival-time or internal-time) is added to the record.

Table 2.2 summarizes the terminology and synonyms from the literature. In the following sections, we define records, streams, and tables as well as time- and ordering-semantics formally.

2.4.1 Records, Streams, and Tables

Our model comprises two entities, namely *streams* and *tables*, which are both composed of *records*. Following the relational model, records, streams, and tables are strongly typed with a *schema* that is defined over a set of domains.

Definition 5 (Schema)**.** *A* schema *is a set of attributes* $\{A_1, \ldots, A_n\}$ *where each attribute is a pair of the form* $\langle name, type \rangle \in \texttt{String} \times \mathbb{D}_{A_i}$*. The first element of an attribute is its name that must be unique within the schema. The second element of an attribute is its domain (or data type).*

Definition 5 is a generic schema definition as used in the relational model. For our stream and table schemas we apply some restrictions:

Definition 6 (Stream Schema)**.** *A* stream schema $\mathbb{S}$ *consists of a set of four attributes* $\{O, T, K, V\}$ *with two special attributes, the offset* $O = \langle \texttt{offset}, \mathbb{N}_0 \rangle$ *and the timestamp* $T = \langle \texttt{timestamp}, \mathcal{T} \rangle$*. The key (*$K$*) serves as an identifier and the actual record payload is encoded in the value (*V*) with* $K = \langle \texttt{key}, \mathbb{D}_K \rangle$ *and* $V = \langle \texttt{value}, \mathbb{D}_V \rangle$*.*

Similarly, a table schema is defined as follows:

Definition 7 (Table Schema)**.** *A* table schema $\mathbb{T}$ *consists of a set of three attributes* $\{T, K, V\}$ *with one special attribute, the timestamp* $T = \langle \texttt{timestamp}, \mathcal{T} \rangle$*. The key (*$K$*) serves as a unique identifier and the actual record payload is encoded in the value (*V*) with* $K = \langle \texttt{key}, \mathbb{D}_K \rangle$ *and* $V = \langle \texttt{value}, \mathbb{D}_V \rangle$*.*

It is important to note that both schema definitions are similar, with the difference that there is no offset attribute in a table schema.[9] Finally, we denote a record schema as $\mathbb{R} = \{T, K, V\}$.

[9] It is important to note that the key-value-pair data model for the record payload may easily be extended to a n-ary tuple based model with a "payload" schema $\{A_1, \ldots, A_n\}$ and a set of key-attributes $K = \{K_1, \ldots, K_n\} \subseteq \{A_1, \ldots, A_n\}$.

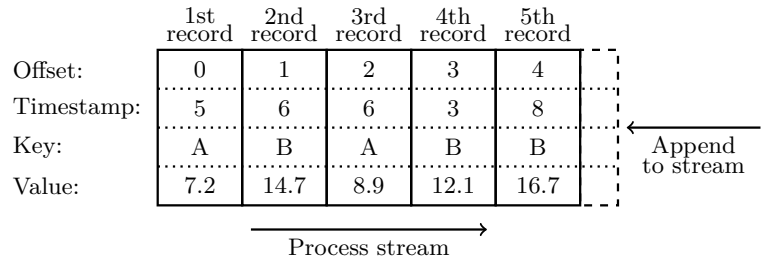

Figure 2.5: Example stream with five records.

Definition 8 (Schema Compatibility)**.** *A record schema* $\mathbb{R}$*, a stream schema* $\mathbb{S}$*, and a table schema* $\mathbb{T}$ *are* compatible *to each other, iff:*

$$\mathbb{R} = \mathbb{S} \backslash O = \mathbb{T} \tag{2.1}$$

Records with schema $\mathbb{R}$ have the following properties:

Definition 9 (Record)**.** *A* record r *with schema* $\mathbb{R}$ *is a triple of the form* $r = \langle t, k, v \rangle$*, with* $[\![\mathbb{R}]\!] = \mathcal{T} \times \mathbb{D}_K \times \mathbb{D}_V$ *being the domain of all records with schema* $\mathbb{R}$*, i. e.,* $\forall r : r \in [\![\mathbb{R}]\!]$*. We denote the record's timestamp, key, an value as* $r.t$*,* $r.k$*, and* $r.v$*.*

Following Definition 9, *stream records* and *table records* are defined analogously based on stream and table schemas. We denote a stream record's offset attribute as $r.o$. If it is clear from the context, we simple use the term record instead of stream or table record in the remainder of this thesis.

Using the definition of schemas and records from above we define *streams* and *tables* as follows:

Definition 10 (Stream)**.** *A* stream S *with schema* $\mathbb{S}$ *(denoted by* $S[\mathbb{S}]$*) is an append-only sequence of immutable records* $r \in [\![\mathbb{S}]\!]$*. We use* $S[\![\mathbb{S}]\!]$ *to denote the domain of all streams with schema* $\mathbb{S}$*. Each record of a stream has a unique offset that is its position within the sequence. Offsets start at zero and are incremented by one for each appended record. We denote a stream with records* $r_0, r_1, r_2 \in [\![\mathbb{S}]\!]$ *as:*

$$S[\mathbb{S}] = (r_0, r_1, r_2) \tag{2.2}$$

The index n of a record r_n indicates the record offset in S, i. e., $r_n.o = n$. Thus, given a record r_o as $\langle o, t, k, v \rangle$, we simplify the notation to $r_o = \langle t, k, v \rangle$ omitting the redundant offset attribute. We allow multiple records within a stream to have the same timestamp, but require that the number of records with the same timestamp is finite (even if it can be arbitrarily large).

An example stream is depicted in Figure 2.5. The respective stream consists of five records, with $\mathbb{D}_K = \{A, B\}$ as the domain of keys and $\mathbb{D}_V = \mathbb{Q}$ as the domain of values. For instance, the second record in the stream is given as $\langle 1, 6, B, 14.7 \rangle$. It is important to note that records are enumerated by their offsets and that new records are appended to the stream on the right hand side with increasing offsets (c. f. Definition 12 of `app` operator below).

The second entity in our model are *tables*, and we define them as follows:

Definition 11 (Table)**.** *A* table T *with schema* $\mathbb{T}$ *(denoted by* $T[\mathbb{T}]$*) is a set of records* $r \in [\![\mathbb{T}]\!]$*. We use* $T[\![\mathbb{T}]\!]$ *to denote the domain of all tables with schema* $\mathbb{T}$*. We denote a table with records* $r_0, r_1, r_2 \in [\![\mathbb{T}]\!]$ *as:*

$$T[\mathbb{T}] = \{r_0, r_1, r_2\} \tag{2.3}$$

Similar to the relational model, the key attribute has primary key semantics, i. e., must be unique over all records in T*:*

$$\forall r \in T, \forall \bar{r} \in T : (r.k = \bar{r}.k \implies r = \bar{r})$$

Our table definition is the same as in the relational model. Therefore, we use standard relational algebra notation in the remainder of this thesis.

The definitions of records, streams, and tables from above all include a timestamp attribute to support event-time semantics. As noted in previous work [SW04] event-time support is a key feature to support deterministic semantics with a mathematical foundation. We assume that the associated event timestamp is embedded into each record by the producer. To this end, our streaming model also defines our storage and processing model. Data stream records are stored in offset order and a stream processor uses a single linear scan over the streams to process the data. When reaching the end of a stream, processing pauses (but does not terminate) until new data is appended to the stream. This assumption is important because it implies that records are processed in offset order in our model. We discuss details about order and time in Section 2.4.4. In the next sections, we introduce auxiliary operations on streams (Section 2.4.2) and tables (Section 2.4.3) that we use to define our operator semantics in Chapter 5.

2.4.2 Stream Operations

We introduce basic stream operations in this section that we use to define the semantics of our processing operators in Chapter 5. These operations are not part of our model itself, but auxiliary operators that simplify the definition of the actual operators.

If a new stream is created, it is the empty sequence denoted by $S[\mathbb{S}] = ()$. New records are written into a stream by appending them to the end of the sequence. We define the *append* operator `app` for streams and records as follows:

Definition 12 (Append Record Operator)**.** *Given a stream* $S[\mathbb{S}] = (r_0, \ldots, r_n)$ *and a record* r *with compatible schema* $\mathbb{R}$*,* $\texttt{app} : S[\![\mathbb{S}]\!] \times [\![\mathbb{R}]\!] \to S[\![\mathbb{S}]\!]$ *appends* r *to* $S[\mathbb{S}]$ *as:*

$$\texttt{app}(S, r) : S = (r_0, \ldots, r_n, r_{n+1}) \tag{2.4}$$

with

$$r_{n+1} = \langle n + 1, r.t, r.k, r.v \rangle$$

It is important to note that `app` modifies the provided input data stream. Furthermore, the input record has no offset attribute and the offset is assigned by `app`.

The *length* operator `length` returns the number of records in a stream. If a stream is infinite, `length` returns ∞.

Definition 13 (Length Operator)**.** *Given a stream S with schema $\mathbb{S}$,* `length` : $S[\![\mathbb{S}]\!] \to \mathbb{N}_0 \cup \{\infty\}$ *returns the number of records contained in S.*

$$\texttt{length}(S) = \begin{cases} 0 & \text{if } S = () \\ \infty & S \text{ is infinite} \\ \max\{r.o | \forall r \in S\} + 1 & \text{otherwise} \end{cases} \tag{2.5}$$

We denote `length`(S) *shortly as* $|S|$*.*

The *first* operator `fst` returns the first record of a stream.

Definition 14 (First Operator)**.** *Given a stream S with schema $\mathbb{S}$,* `fst` : $S[\![\mathbb{S}]\!] \to [\![\mathbb{S}]\!]$ *returns the record with offset zero in S.*

$$\texttt{fst}(S) = r_0 \tag{2.6}$$

with

$$r_0 \in S$$

The `fst` operator is not defined for the empty sequence and it is invalid to call `fst`$(())$.

The *prefix* operator `pre` returns the first n records of a stream. It is important to note that `fst` is not a special case of `pre` with $n = 1$ as it returns a single record while `pre` would return a sequence with one record.

Definition 15 (Prefix Operator)**.** *Given a stream S with schema $\mathbb{S}$ and a number $n \in \mathbb{N}_0$,* `pre` : $S[\![\mathbb{S}]\!] \times \mathbb{N}_0 \to S[\![\mathbb{S}]\!]$ *returns a finite substream of S with n records starting at offset zero in S.*

$$\texttt{pre}(S, n) = (r_0, \ldots, r_{n-1}) \tag{2.7}$$

with

$$\forall r_i \in \texttt{pre}(S, n) : r_i \in S$$

If n is zero, `pre` returns the empty sequence. The result of `pre` is shorter than n if $n >$ `length`(S). In particular, if $n \geq$ `length`(S) then `pre`$(S, n) = S$.

The *suffix* operator `sfx` returns the tail of a stream starting at a specified offset. It is important to note that `sfx` returns an infinite stream if the input stream is infinite. Furthermore, in our model all streams are append-only sequences and start with offset zero. Thus, the result streams also starts with offset zero—it is a new stream, i. e., a partial copy of the input stream.

Definition 16 (Suffix Operator)**.** *Given a stream S with schema $\mathbb{S}$ and a number $n \in \mathbb{N}_0$,* `sfx` : $S[\![\mathbb{S}]\!] \times \mathbb{N}_0 \to S[\![\mathbb{S}]\!]$ *returns a substream of S beginning at offset n.*

$$\texttt{sfx}(S, n) = (\bar{r}_0, \ldots, \bar{r}_{\texttt{length}(S)-n-1}) \tag{2.8}$$

with

$$\forall \bar{r} \in \texttt{sfx}(S, n) : (\bar{r} = \langle o, t, k, v \rangle \wedge \exists r \in S : r = \langle o + n, t, k, v \rangle)$$

It is important to note that if $n \geq$ `length`(S), the result of `sfx` is the empty sequence.

Similar to appending a record to a stream (Definition 12) we define an *append stream* operator `app` that concatenates two streams. We denote both append operators as `app` as it is clear from the context which one is used.

Definition 17 (Append Stream Operator)**.** *Given a* finite *stream S_1 and a potentially infinite stream S_2 both with the same schema $\mathbb{S}$,* $\texttt{app} : S[\![\mathbb{S}]\!] \times S[\![\mathbb{S}]\!] \to S[\![\mathbb{S}]\!]$ *returns a stream of records that is the concatenation of both input streams.*

$$\texttt{app}(S_1, S_2) = (\bar{r}_0, \ldots, \bar{r}_{n-1}, \bar{r}_n, \ldots, \bar{r}_{n+m-1}) \tag{2.9}$$

with

$$\begin{aligned} n = \texttt{length}(S_1) \ \wedge \\ m = \texttt{length}(S_2) \ \wedge \\ \forall \bar{r} \in \texttt{app}(S_1, S_2) : (\bar{r} = \langle o, t, k, v\rangle \ \wedge \\ (o < n \implies \exists r \in S_1 : \bar{r} = r) \ \wedge \\ (o \geq n \implies \exists r \in S_2 : r = \langle o - n, t, k, v\rangle)) \end{aligned}$$

In this section, we defined auxiliary operators on streams. In the next section, we define similar operators on tables.

2.4.3 Table Operations

We introduce basic table operations in this section that we use to define the semantics of our processing operators in Chapter 5. Those operations are not part of our model itself, but auxiliary operators that simplify the definition of the actual operators.

The *lookup* operator `lookup` takes a table and a primary key as input and returns the corresponding record from the table.

Definition 18 (Lookup Operator)**.** *Given a table T with schema $\mathbb{T}$ and a lookup key $k \in \mathbb{D}_K$,* $\texttt{lookup} : T[\![\mathbb{T}]\!] \times \mathbb{D}_K \to [\![\mathbb{T}]\!] \times \{\bot\}$ *is defined as:*

$$\texttt{lookup}(T, k) = \begin{cases} r & \text{if } \exists r \in T : r.k = k \\ \bot & \text{otherwise} \end{cases} \tag{2.10}$$

We denote $\texttt{lookup}(T, k)$ *also as* $T.\texttt{lookup}(k)$

It is important to note that `lookup` is not the same as the relational selection operator because it returns a record instead of a set of records.

The *insert* operator `insert` takes a table and a record as input and returns a modified table containing the input record.

Definition 19 (Insert Operator)**.** *Given a table T with schema $\mathbb{T}$ and a record r with compatible schema $\mathbb{R}$,* $\texttt{insert} : T[\![\mathbb{T}]\!] \times [\![\mathbb{R}]\!] \to T[\![\mathbb{T}]\!]$ *is defined as:*

$$\texttt{insert}(T, r) : T = T \backslash \{\bar{r} \in T | \bar{r}.k = r.k\} \cup \{r\} \tag{2.11}$$

We denote $\texttt{insert}(T, r)$ *also as* $T.\texttt{insert}(r)$

It is important to note that `insert` modifies the provided input table. Furthermore, it preserves primary key semantics by removing a potentially existing record with the same key as the new record from the table first.

The *delete* operator `delete` takes a table and a key $k \in \mathbb{D}_K$ and returns a modified table that does not contain a record with the input key.

Definition 20 (Delete Operator)**.** *Given a table T with schema $\mathbb{T}$ and a key $k \in \mathbb{D}_K$,* $\texttt{delete} : T[\![\mathbb{T}]\!] \times \mathbb{D}_K \to T[\![\mathbb{T}]\!]$ *is defined as:*

$$\texttt{delete}(T, k) : T = T \backslash \{\bar{r} \in T | \bar{r}.k = k\} \tag{2.12}$$

We denote $\texttt{delete}(T, k)$ *also as* $T.\texttt{delete}(k)$

It is important to note that `delete` modifies the provided input table.

2.4.4 Order and Time

In stream processing, operator semantics are based on timestamp order. For the general case, it is only possible to define a *deterministic* model if input data is *strictly* ordered. In our model, streams are sequences of records (Definition 10) that are strictly ordered by their offsets, called *offset order*. However, timestamps are not guaranteed to be unique, and thus, the time domain does *not* induce a strict total timestamp order. Hence, we define *time-based record order* as follows:

Definition 21 (Time-based Record Order)**.** *Given two records r and $\bar{r}$ from a stream S. Record r is* earlier *(or* before*) record $\bar{r}$, denoted $r < \bar{r}$, iff:*

$$r.t < \bar{r}.t \vee (r.t = \bar{r}.t \wedge r.o < \bar{r}.o)$$

In this case, we also say that $\bar{r}$ is later *(or* after*) r.*

Definition 21 uses the record offset as the "tie breaker" if records have the same timestamp [JMS+08]. Since order is defined based on event-timestamps, but records are stored in offset order, we introduce the notion of *out-of-order* records as follows:

Definition 22 (Out-of-Order Records)**.** *A record $r \in S$ is* out-of-order *iff:*

$$\exists \bar{r} \in S : (\bar{r}.o < r.o \wedge \bar{r}.t > r.t)$$

We say that a stream is *ordered* if no out-of-order records exist in the stream; otherwise the stream is *unordered*. Ordered data streams are not guaranteed in our model because we use event-time semantics. We discuss how out-of-order data is handled in Chapter 5.

For some cases, we are not interested in the order over all records, but only in the order of all records with the same key. We define the notion of out-of-order records *with respect to the key* as follows:

Definition 23 (Key-based Out-of-Order Records)**.** *A record $r \in S$ is* out-of-order with respect to its key *k iff:*

$$\exists \bar{r} \in S : (\bar{r}.k = r.k \wedge \bar{r}.o < r.o \wedge \bar{r}.t > r.t)$$

2.5 Related Work

Data stream processing was in the focus of the database community for the first time since the early 2000s resulting in a large variety of research prototypes. Most of those system are centralized and single threaded, and address fundamental question in data stream processing.

Some of the first systems specialize on certain use cases instead of general purpose stream processing. For example, Tribeca [Sul96, SH98] is a single input in-memory stream processing system for network traffic analysis. It supports filters, projections, windowed aggregations, and splitting/merging of data streams. The NiagaraCQ system [CDTW00, KNV03, VN02] is a continuous processing extension to the Niagara distributed XML database system. The research goal was to detect common sub-queries of continuous queries and merge new queries into existing ones to share partial results. Gigascope [CGJ+02, CJSS03b, CJSS03a, JMSS05] is a specialized stream processor for network monitoring and analytics. It has a two-tier architecture to evaluate sub-queries at the data sources for data reduction, before sending the data streams in the centralized system. It uses a SQL-like, but quite limited and specialized, query language GSQL and exploits *ordering properties* for query processing. Additionally to system architecture and design, research in approximate aggregations and sampling was conducted [CJK+04, JMR05].

The TelegraphCQ project [MF02, MSHR02, CF02, KCC+03, CCD+03b, CCD+03a, SHCF03, RSW+07] focused on adaptive data stream processing techniques. Because stream processing programs are long running continuous queries, and input data streams are not known in advance, there are no available statistics that could be used for query optimization. Furthermore, data characteristics may change over time, and thus, a currently optimal query execution plan may be sub-optimal later. Hence, queries need to be optimized on-the-fly and changed adaptively during runtime. TelegraphCQ proposes the highly adaptive Eddy operator [AH00, Des04] and so-called State-Modules [MSHR02, RDH03] to build an adaptive query processing system.

Other research focuses on general purpose stream processing. The Stanford Stream Data Manager (STREAM) project [BW01, BBD+02, ABB+03a, ABB+03b, MWA+03, BW04] developed a centralized system that uses query plans consisting of operators, queues, and synopsis (to store state). STREAM is inspired by relational database technology and pioneered general purpose SQL-like data stream processing (in contrast to Gigascope). Queries are expressed with a declarative SQL-like query language called CQL (Continuous Query Language) [ABW03, AW04, ABW06] that transforms data streams into tables to execute the specified operators using relational semantics. We discuss more details about CQL in the second part of this thesis (Chapter 5). Furthermore, the project investigated (sub-)query sharing, result approximation, and scheduling [BBMD03, BBD+04].

The Aurora project [CcC+02, ACc+03b, ACc+03a, CcR+03, BBC+04] was the first to focus on data flows to express stream processing programs ("box and arrow model"). Additionally to continuous queries, ad-hoc queries are also supported. To reason about queries and to allow for query optimization similar to relational database systems, Aurora introduces its own Stream Query Algebra (SQuAl) in-

cluding window operators, and describes detailed operator scheduling techniques. A novel concept in Aurora is the notion of Quality of Service (QoS) that allows to trade-off response-time, result completeness, and result utility via *load shedding*. Related to load-shedding is Aurora's runtime cost model that incorporates a notion of throughput and capacity that is similar to—but more basic than—our cost model (Chapter 3).

Aurora* and Medusa [CBB+03, ZSC+03, BBS04] are distributed and federated stream processing prototypes based on Aurora. Later Borealis, an advancement of Aurora, Aurora*, and Medusa, was developed [AAB+05, XZH05, ABC+05, RMCZ06, BBMS08, HCCZ08]. Borealis is able to distribute the load of queries over multiple nodes in a processing cluster, considering the structure of the data flow program, operator load, and node utilization. Because Borealis is distributed, fault-tolerance is considered, too, and a novel approach called *Delay, Process, and Correct* (DPC) [BBMS05] that allows users to trade-off availability vs. consistency is proposed. Furthermore, Borealis extends Aurora's data model by allowing for insert, delete, and replacement messages, and adds control messages that can be used to updated filter predicates or window sizes at runtime.

System S is a distributed stream processing system [AJS+06, AAB+06, JAA+06, GAW+08] using an adaptive resource control algorithm and a declarative language to express streaming data flow programs. Its SODA optimizer [WBH+08] implements an adaptive scheduling algorithm with the goal to maximize the result utility and balancing load. The assumption is that the input data rate it too high and not all records can be processed in a timely manner. System S also supports a limited form of data-parallel processing [WKWO12]. Instead of sharding state over multiple nodes, multiple thread use a shared state to process input records in parallel.

The latest generation of stream processing systems focus on data-parallel processing and horizontal scaling inspired by MapReduce [DG04, Dea06, DG08]. In this thesis, we investigate the runtime behavior of such systems.

StreamCloud [GJPPMV10, GJPPM+12] is a research prototype built on top of Borealis. It proposed a novel protocol for load balancing and dynamic scaling, as well as a fault-tolerance protocol including operator state. StreamCloud also introduces multiple parallelization strategies that are compared to each other based on a cost model.

Nephele streaming [LWK12, LWK14, LJK15] is similar to StreamCloud, however the research focus is on Quality of Service guarantees. The impact of dynamic batching, dynamic task chaining, and dynamic scaling on the processing latency in streaming data flows is studied. The goal of the proposed optimization algorithms is to meet latency constraints.

Apache S4 [ASFd, NRNK10] is inspired by MapReduce and SPC [AAB+06]. While it offers only a low level programming interface and only supports ephemeral in-memory operator state, it is one of the first systems used in the industry. Muppet [LLP+12] is similar to S4, however, allows to backup operator state in an external key-value store to provide improved fault-tolerance. Apache Storm [ASFf, TTS+14] and Apache Heron [ASFb, KBF+15] (a successor of Storm) offer similar programming abstractions as S4 and Muppet. Both are noteworthy, because they are the first systems that gained a broader adoption in industry, and inspired many research

papers [ABQ13, NMG+15, YM15, PHH+15, CDE+16, XPG16, FAG+17]. We also use Apache Storm in this thesis.

MapReduce inspired systems like S4, Muppet, Storm, and Heron do not support sophisticated state handling or fault-tolerant state. Those limitations are addressed by Apache Samza [ASFe, KK15, NPP+17], SEEP [CFMKP13], and Apache Flink [ASFa, CKE+15, CEF+17]. All three systems use operator local state instead of an external data system for high performance state access. While Samza materializes a write-ahead-log (WAL) into an external system for fault-tolerance, SEEP and Flink checkpoint operator state. SEEP integrates checkpointing with dynamic stateful scaling. Flink focuses on globally consistent checkpoints, implementing a distributed snapshot algorithm based on Chandy-Lamport [CL85].

Orthogonal to distributed systems research, novel stream processing programming models and semantic models have been developed as well. Akidau et al. [ABB+13, ABC+15] suggest a watermark and trigger based evaluation model, to handle out-of-order data and to allow users to trade-off correctness vs. latency and cost. Trill [CGB+14] introduces a temporal-relational processing model that combines real-time stream processing, temporal historical, and offline processing. To control processing latencies for the real-time streaming case, punctuations are used.

Part II

Cost-based Streaming Data Flow Optimization

Chapter 3

Streaming Data Flow Cost Model

Contents

This chapter introduces our cost model for streaming data flow programs that considers processing and network requirements. Our model is used to optimize data flow provisioning and therefore we define processing cost to include CPU utilization as well as disk access. Main-memory is excluded from the cost model because modern distributed stream processing systems either use external storage to preserve state [BROL14, CDE+16] or use local disk-based state management [CEF+17]. Thus, we assume that available storage is much larger than required—i. e., no main-memory limitation—, and hence memory is no provisioning concern.

Our cost model is designed with the notion of *capacity* (formally defined in Section 3.1) as core building block. The main idea is to model how much data a data flow program—or to be more precise, an execution graph—can process. Given a target input data rate, we use the cost model to compute a deployment configuration that avoids runtime bottlenecks. While we introduce the cost model itself in this chapter, we discuss its application in Chapter 4.

The capacity of a data flow program and its operators depends on the CPU and network costs (Section 3.2 and Section 3.3). For CPU costs, we include batching as introduced in Section 2.1 in our model that allows us to reduce the required resources to execute a data flow program. In particular, we describe different batching techniques for stream processing in detail in Section 3.4. For the overall cost

model, we first focus on single operator costs in isolation. In a second step, we extend our model to data flow graphs and include upstream/downstream operator dependencies. Hence, we consider the structure of the data flow program in our model allowing for performance prediction for complex data flows. At the end of this chapter, we evaluate our cost model experimentally using a processing cluster to show that our cost model reflects real-world observations.

3.1 Data Flow Capacity

In order to address the cost model requirements as described in Section 2.1.2 we use the notion of *capacity* as the central concept of our cost model. We define capacity as a generic concept in an abstract way as follows:

Definition 24 (Capacity)**.** *The* capacity *is the maximum amount of work that can be performed in a certain time span.*

The *capacity* concept allows us to model cost per time unit. As discussed in Section 2.1.2, modeling absolute costs would not be useful for an infinite incoming data stream, because costs would be infinite [VN02].

For our cost model we apply the general capacity concept (Definition 24) to data flow programs as well as to individual operators. Because data flow programs consist of operators, the individual operator capacities determine the capacity of the overall data flow program. Before we can illustrate the relationship between operator and data flow capacity, we formally define the capacity of a data flow program as follows:

Definition 25 (Data Flow Capacity)**.** *The* capacity of a data flow program *is the maximum number of input records per input stream that a data flow program can process per time unit. Given a data flow program D with sources $S = \langle s_1, \ldots, s_n \rangle$, the capacity $\mathbb{C}$ of D is:*

$$\mathbb{C}(D) = \langle c_1, \ldots, c_n \rangle \tag{3.1}$$

where c_i denotes the maximum number of input records per time unit for s_i such that the data flow program D is able to process all input streams simultaneously.

The capacity of a data flow program is a vector that describes the maximum throughput per source a data flow program can process. If it is clear from the context to which data flow program D a capacity refers we denote $\mathbb{C}$ instead of $\mathbb{C}(D)$ in the remainder of this work. Modeling the capacity as a vector has two implication (c. f. Example 3 below):

1. $\mathbb{C}$ is not unique for D, i. e., there are multiple different data flow capacities for a data flow program D. We denote the set of all $\mathbb{C}$ of D as $\mathbb{C}^\dagger(D)$.

2. There are dependencies between all values c_i, and all c_i *together* must meet the criteria of "maximum number of input records [...] D is able to process" with respect to the complete data flow program. Hence, the term *"maximum number of input records per time unit for s_i"* cannot be applied in isolation to each source to define the capacity of D (c. f. Example 3). These dependencies are implicitly expressed in Definition 25 as *"such that [...] D is able to process all input streams simultaneously"*.

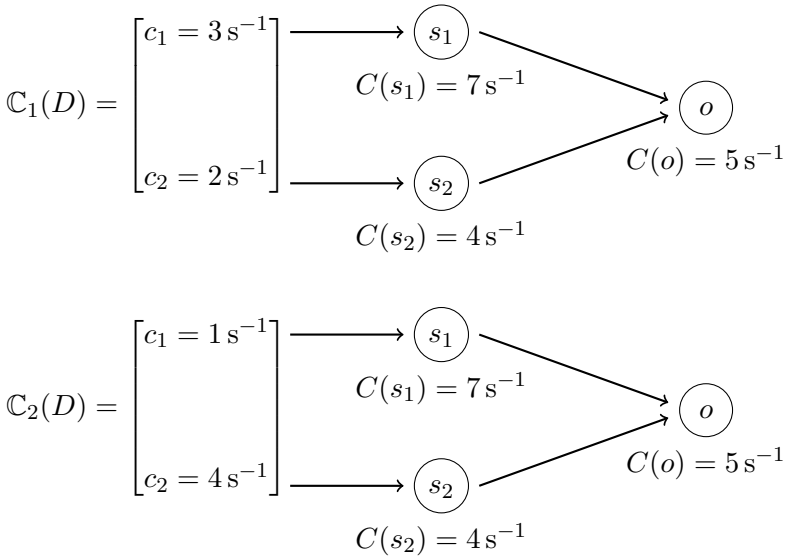

Figure 3.1: Data flow program with three nodes having multiple data flow capacities $\mathbb{C}_1(D)$ and $\mathbb{C}_2(D)$.

We use Example 3 to discuss both implications in more detail.

Example 3. *Assume a data flow program with two source operators (s_1 and s_2) and a single sink operator (o) that is connected to both sources as shown in Figure 3.1. Furthermore, let the operator capacities be the following: $C(s_1) = 7\,\mathrm{s}^{-1}$ (i. e., s_i can process 7 records per second), $C(s_2) = 4\,\mathrm{s}^{-1}$, and $C(o) = 5\,\mathrm{s}^{-1}$. Since the sink is connected to both sources, it processes all incoming data records of both sources simultaneously. As the sink can process maximum 5 records per second, the sum of both output data rates of the sources cannot be larger than 5 records per second. Thus, some example capacities of the data flow program are $\mathbb{C}_1 = \langle 3\,\mathrm{s}^{-1}, 2\,\mathrm{s}^{-1}\rangle$, $\mathbb{C}_2 = \langle 1\,\mathrm{s}^{-1}, 4\,\mathrm{s}^{-1}\rangle$, $\mathbb{C}_3 = \langle 5\,\mathrm{s}^{-1}, 0\,\mathrm{s}^{-1}\rangle$, etc.*

Counter examples that exceeds the capacity of D are $\langle 9\,\mathrm{s}^{-1}, 2\,\mathrm{s}^{-1}\rangle$, $\langle 1\,\mathrm{s}^{-1}, 7\,\mathrm{s}^{-1}\rangle$, $\langle 6\,\mathrm{s}^{-1}, 0\,\mathrm{s}^{-1}\rangle$ or $\langle 4\,\mathrm{s}^{-1}, 3\,\mathrm{s}^{-1}\rangle$: in the first two examples, the capacity of source s_1 or s_2 is exceeded, respectively, violating Definition 25. For the last two examples, both sources could handle a corresponding input data rate themselves, however, the sink cannot handle an input data rate of more than 5 records per second, and thus, both examples exceed the capacity of D, too.

Example 3 shows that $\mathbb{C}$ is not unique for D—it lists three concrete capacities for D (not comprehensive). Additionally, the example illustrates the dependencies between all c_i (Item 2) from Definition 25 and that the maximum capacity per source cannot be used to define the data flow capacity. Formally, we define the maximum capacity for sources as follows:

Definition 26 (Maximum Source Capacity)**.** *Given the set of all capacities $\mathbb{C}^\dagger$ for a data flow program D with sources $S = \langle s_1, \ldots, s_n\rangle$, the* maximum capacity $\widehat{c}_i$ *for source s_i is defined as:*

$$\widehat{c}_i = \max\{c_i | \langle c_1, \ldots, c_i, \ldots, c_n\rangle \in \mathbb{C}^\dagger\} \tag{3.2}$$

The maximum capacity $\widehat{c}_i$ is the maximum input data rate for source s_i the data flow program D can possibly process.

Example 3 (cont.). *For D from Figure 3.1 we can infer that $\widehat{c}_1 = 5\,\mathrm{s}^{-1}$ and $\widehat{c}_2 = 4\,\mathrm{s}^{-1}$. We emphasize that the maximum capacity $\widehat{c}_1$ for source s_1 is smaller than the capacity of s_1 (i. e., $\widehat{c}_1 = 5\,\mathrm{s}^{-1} < 7\,\mathrm{s}^{-1} = C(s_i)$), because the capacity of node o limits the maximum throughput for source s_1 in the overall data flow program. For $\widehat{c}_2$, the capacity of s_2 itself is the limiting factor.*

However, $\widehat{c}_i$ cannot be used to compute $\mathbb{C}$ as we will show in the following:

Example 3 (cont.). *Both maximum source capacities $\widehat{c}_1$ and $\widehat{c}_2$ together, i. e., $\langle \widehat{c}_1, \widehat{c}_2 \rangle = \langle 5\,\mathrm{s}^{-1}, 4\,\mathrm{s}^{-1} \rangle$, exceed the capacity of D, because the downstream node o has a capacity that is smaller than the sum of both source capacities (i. e., $C(p) = 5\,\mathrm{s}^{-1} < 9\,\mathrm{s}^{-1} = 5\,\mathrm{s}^{-1} + 4\,\mathrm{s}^{-1} = \widehat{c}_1 + \widehat{c}_2$).*

Hence, the term *"maximum number of input records per time unit for s_i"* in Definition 25 does not imply that $c_i = \widehat{c}_i$. Definition 25 excludes $\langle \widehat{c}_1, \widehat{c}_2 \rangle$ from the capacity definition via *"such that the data flow D is able to process all input streams simultaneously."*. We point out that $\langle \widehat{c}_1, \widehat{c}_2 \rangle \notin \mathbb{C}^\dagger$ in our concrete example. In general, $\langle \widehat{c}_1, \cdots, \widehat{c}_n \rangle$ may be a capacity for D though.

Example 3 illustrates that the data flow capacity $\mathbb{C}$ depends on the individual operator capacities. Next, we defined *operator capacity* (Definition 27) based on *task capacity* (Definition 28) that we define afterwards. Data flow programs are translated into execution graphs, and thus, we distinguish between the capacity of an operator and the capacity of a task. Because we assume linear scaling due do data parallelism, the capacity of an operator is the capacity of a task multiplied by the number of operator tasks, i. e., the degree of parallelism of the operator.

Definition 27 (Operator Capacity). *Given a operator $v \in V$ of a data flow program $D = (V, E)$ with tasks $t^v_1, \ldots, t^v_{dop(v)}$. We assume that each task has the same capacity $C(t^v)$, i. e., $C(t^v) = C(t^v_1) = \cdots = C(t^v_{dop(v)})$. Thus, the* operator capacity $C(v)$ *of v is:*

$$C(v) = dop \cdot C(t^v) \tag{3.3}$$

If it is clear from the context if we refer to an operator capacity or a task capacity, we denote C instead of $C(v)$ or $C(t^v)$ in the remainder of this work. The *task capacity* is defined as follows:

Definition 28 (Task Capacity). *The* capacity of a task *is the maximum number of input records per time unit a task can process. It is determined by three parameters: its* input capacity C_i, *its* processing capacity C_p, *and its* output capacity C_o. *We define the task capacity C as:*

$$C = \min\{C_\mathrm{i}, C_\mathrm{p}, C_\mathrm{o}\} \tag{3.4}$$

C_i and C_o model network input and output while C_p models CPU costs. Hence, the individual parameters C_i, C_p, and C_o are independent from each other and depend on the operator itself as well as on the execution environment, i. e., the underlying hardware. We take the minimum over all three capacities, because one

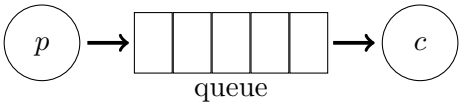

Figure 3.2: Data exchange via a queue between tasks of two operators.

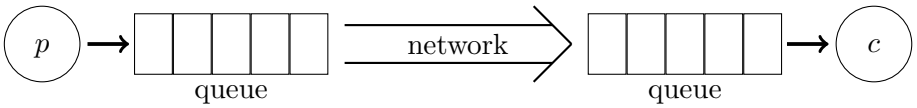

Figure 3.3: Data exchange via queues and network between tasks of two operators.

of them will be the limiting factor for the overall task capacity. All three values are defined with respect to the maximum number of *input* records per time unit a task can process. In particular, the *output* capacity, is defined as the maximum *input* data rate, such that the resulting output data rate does not result in a network bottleneck. For sources, we do not compute an input capacity but set $C_\mathrm{i} = \infty$. Accordingly for sinks, we do not compute an output capacity but set $C_\mathrm{o} = \infty$. This allows us to use Equation 3.4 for all tasks without the need to distinguish different task types. We formally introduce and define the processing capacity C_p, as well as network capacities C_i and C_o in the following sections.

3.2 Processing Costs

Based on our general definition of capacity (Definition 24), we formally define the *processing capacity* of a task as follows (c. f. [CcC+02]):

Definition 29 (Processing Capacity)**.** *The* processing capacity C_p *of a task is the* maximum *number of records the task can process per time unit. Let* c_p *be the* processing cost *for a single input record; processing costs are defined in time units. We define the* processing capacity *of a task as:*

$$C_\mathrm{p} = \frac{1}{c_\mathrm{p}} \tag{3.5}$$

Because we define the *processing costs* as number of time units per record, the processing capacity C_p of a task is the inverse of c_p (c. f. [CcC+02]). Before breaking down the processing costs and introducing our processing cost model for tasks, we consider some more details of the execution model. Data stream processing programs are modeled as data flow programs as discussed in Chapter 2. Tasks of different operators are connected with queues [BBD+02, CcC+02] to decouple them and to allow them to be executed by different threads [MF02, LLP+12, LWK12]. An upstream task puts its output records into a queue, while the downstream task pulls its input records from a queue (Figure 3.2). Similarly, if tasks are executed on different physical machines, there are additional network threads [TTS+14, KBF+15] that are responsible to send and receive data (Figure 3.3). Thus, threads that execute a task exchange data with network threads via queues, too.

As indicated in Definition 29, the core unit of our cost model is processing costs of a single record. Additionally, we discussed that tasks exchange data via queues.

Hence, the processing costs consist of three parameters: cost to process a single record as well as the cost to enqueue and dequeue records. Because there is *no* one-to-one relationship between the number of input and output records[1] we also need to consider the operator selectivity (denoted by s). We model the selectivity as average number of output records per input record [ACc+03b]. We point out that the selectivity can be larger than 1 for this definition [BBMD03].

Definition 30 (Record Processing Cost)**.** *The* processing costs *of a task are dominated by three parameters. Cost to dequeue one input record (c_{fetch}) from the input queue, cost to process a single record (c_{cpu}), and cost to enqueue output records (c_{emit}) into the output queue (this cost might apply multiple times depending on the operator selectivity s). We model all of those costs as time units spent to perform the corresponding action for a single record. We define the processing costs c_{p} for a single record of a task as (c. f. [CcC+02]):*

$$c_{\text{p}} = c_{\text{fetch}} + c_{\text{cpu}} + s \cdot c_{\text{emit}} \tag{3.6}$$

We set $c_{\text{fetch}} = 0$ for sources that do not have incoming edges. For sinks we set $s = c_{\text{emit}} = 0$ because they have no outgoing edges. This allows us to handle all tasks the same way without the need to distinguish different task types.

3.2.1 Improvements of Throughput with Batching

Data transfer between tasks via queues may result in high CPU and/or network overhead [LWK12]. As tasks run on different threads, the queues between tasks must be synchronized to guard against concurrent access. This results in lock contention as both threads must acquire a lock before accessing the queue.

To reduce the queuing overhead, batching can be used to reduce the lock contention between consecutive tasks. Multiple records are collected within a batch, and the whole batch of records is `put()` into the queue at once. On the consumer side, a task receives a batch of records returned by a single `poll()` operation on the queue. Hence, using batching, the number of queue operations is reduced and lock contention is mitigated. Batching has the drawback of an increased processing latency though. If a task produces an output record and inserts it into its output batch, this record cannot be consumed by the downstream task until the batch is full and the whole batch is sent downstream. Thus, to keep latency low, it is desired to keep the batch size as small as possible.

However, it is hard to estimate the required batch size. Thus, in practice, a trial-and-error approach is applied to find appropriate batch sizes for different operators[2] that reduces the overhead, but does not increase latency unnecessary. This is a long and error prone process. Some research was conducted to apply auto-batching techniques, to automate this trial-and-error approach and to dynamically change the batch sizes during runtime [LWK12]. Similar dynamic batch size techniques are also applied to micro-batching systems [DZSS14]. However, those models only follow a reactive approach and lack a holistic cost model. Therefore, we cannot apply them to compute a *configuration*[3] for a data flow program.

[1] An operator could implement a `filter` or `flatMap` function, and can—in general—emit zero, one, or multiple output records per input record.

[2] All tasks of an operator use the same batch size, as configured on the operator.

[3] We formally define a *configuration* in Chapter 4.

Additionally to batching, there are other techniques like operation fusion[4] [CcC+02, CcR+03, CRP+10, HSS+14, LWK12, SGH15] that help to increase throughput. Those techniques are complementary to batching, and thus, we assume that operator fusion is independent of our cost model. That is, fusion may be applied before using our cost model. There are different trade-offs and limitations for operator fusion that we mention for completeness:

- Operator fusion is only possible for record-at-a-time operations (i. e., no data repartitioning/grouping is required between consecutive operators).
- Operator fusion is only possible if two operators are executed on the same machine, which may not be the case in a distributed system.
- Even if operator fusion is possible, it can be better to run each operator individually if
  - both operators are compute intensive and
  - the operators are executed in a multi-core environment and
  - if horizontal scaling is not possible due to a lack of data parallelism.

With the above assumptions we refine our task cost model and put batching into account. Assume we have a batch size of b records. For this case, a task gets b records from the input queue at once. Furthermore, enqueuing only happens after a batch of output records is available. Before that, a task internally buffers output records to assemble a full batch.

We point out that we model input and output batches independently. Modeling both independently is different to other batching techniques [CGB+14] or the micro-batching approach (Section 2.1). Those systems assemble one batch of input records at the source level and forward the batch from operator to operator. Thus, the batch size is fixed for input and output over multiple operators and it is determined by the source operator. Tuple batching in Aurora is fine grained, however, only based on operator input records and used to improve operator scheduling [CcC+02, CcR+03]. Since we model input and output batches independently for a single operator, we allow for different batch sizes, namely input batch size b_{in} and output batch size b_{out}. Hence, there is one batch enqueue operation (with cost c_{emit}) after b_{out} output records were produced.

Definition 31 (Batch Processing Cost)**.** *The* processing costs c_{p}^{b} *of one batch of input records (with batch size* b_{in}*), for a task with selectivity* s *and an output batch size* b_{out}*, is defined as:*

$$c_{\text{p}}^{\text{b}} = c_{\text{fetch}} + b_{\text{in}} \cdot c_{\text{cpu}} + \frac{s \cdot b_{\text{in}}}{b_{\text{out}}} \cdot c_{\text{emit}} \tag{3.7}$$

We infer from Equation 3.7 that the costs for processing one batch of records is the cost of dequeuing one batch of input records, plus the cost to process each of the input records, plus the cost of enqueuing output batches. Enqueuing an output

[4]Operator fusion is also called *combining boxes*, *operator batching*, or *task chaining* in the literature.

batch happens after the number of output records $s \cdot b_{\text{in}}$ exceeds b_{out}. Hence, it is possible that none or multiple enqueue operations are executed per input batch. If no enqueue operation is executed, it implies that multiple input batches must be processed to complete an output batch. We emphasize that Equation 3.7 is an average cost notation per input batch.

Example 4. *Assume an input batch size of* 10 *records* ($b_{\text{in}} = 10$)*, an output batch size of* 20 *records* ($b_{\text{out}} = 20$)*, and an operator selectivity of* 50 % ($s = 0.5$)*. After processing the first batch of records, there are* $s \cdot b_{\text{in}} = 0.5 \cdot 10 = 5$ *output records. In order to complete an output batch of* 20 *records, its required to process* 4 *input batches. Thus, the cost of enqueuing an output batch, is shared over* 4 *input batches, and we get cost of* $0.25 \cdot c_{\text{emit}}$ *per input batch on average.*

To simplify our model, we normalize the cost of processing a batch of input records to the average cost of processing a single input record. This simplification allows us to compare the impact of batching with the base model (i. e., Equation 3.6 and Equation 3.7). We compute the normalized cost c_{p} by dividing the cost to process one input batch by the input batch size as follows:

$$\begin{aligned} c_{\text{p}} &= \frac{c_{\text{p}}^{\text{b}}}{b_{\text{in}}} \\ &= \frac{c_{\text{fetch}} + b_{\text{in}} \cdot c_{\text{cpu}} + \frac{s \cdot b_{\text{in}}}{b_{\text{out}}} \cdot c_{\text{emit}}}{b_{\text{in}}} \\ &= \frac{c_{\text{fetch}}}{b_{\text{in}}} + c_{\text{cpu}} + \frac{s \cdot c_{\text{emit}}}{b_{\text{out}}} \end{aligned} \tag{3.8}$$

Comparing Equation 3.8 and Equation 3.6 yields that Eq. 3.6 is a special case of Eq. 3.8 with $b_{\text{in}} = b_{\text{out}} = 1$. This relationship proves that our model refines our initial base model. Furthermore, we observe that the processing costs have a lower bound, namely c_{cpu}. With an increasing input and output batch size, the dequeuing and enqueue cost can be shared over as many records as desired. Hence, for an infinite input or output batch size, the amortized dequeue and enqueue costs are zero:

$$\lim_{b_{\text{in}} \to \infty} \frac{c_{\text{fetch}}}{b_{\text{in}}} = 0 \quad \text{and} \quad \lim_{b_{\text{out}} \to \infty} \frac{s \cdot c_{\text{emit}}}{b_{\text{out}}} = 0 \tag{3.9}$$

Therefore, the theoretical minimum processing cost is:

$$\min c_{\text{p}} = \lim_{\substack{b_{\text{in}} \to \infty \\ b_{\text{out}} \to \infty}} \frac{c_{\text{fetch}}}{b_{\text{in}}} + c_{\text{cpu}} + \frac{s \cdot c_{\text{emit}}}{b_{\text{out}}} = c_{\text{cpu}} \tag{3.10}$$

Since processing cost has a lower bound, processing capacity (Definition 29) has an upper bound:

$$\widehat{C}_{\text{p}} = \max C_{\text{p}} = \frac{1}{\min c_{\text{p}}} = \frac{1}{c_{\text{cpu}}} \tag{3.11}$$

We discuss the impact of this relationship in more detail in Chapter 4.

3.2.2 Operator Dependencies

In the previous section, we focused on a single tasks of a single operator and discussed that input and output batch size are independent from each other. In this section, we put the focus on inter operator dependencies, or to be more precise, on consecutive operators. For example, if a downstream operator receives data from an upstream operator, the output batch size of the producer is the input batch size of the consumer. Furthermore, if a consumer receives data from multiple producers it has input batches of different sizes in general.

Corollary 1. *Based on the definition of data flow programs, execution graphs, and our cost model, operators can only be configured with output batch sizes. Operator input batch sizes depend on those output batch sizes and cannot be configured.*

Before we discuss the details of operator dependencies, we define input and output data rates for tasks and operators. Additionally, we describe how the output data rate of an operator or task depends on the corresponding input data rate.

Definition 32 (Task Input and Output Data Rate)**.** *Let $t \in T$ be a task in an execution graph $EG = (T, F)$. The* task input data rate $r_{\text{in}}(t)$ *is the number of input record per time unit of t. The* task output data rate $r_{\text{out}}(t)$ *is the number of output record per time unit of t.*

If it is clear from the context to which task r_{in} and r_{out} belongs, we omit the parameter t in the formulas. Similar to task input and output data rate, we define input and output data rate for data flow operators as follows:

Definition 33 (Operator Input and Output Data Rate)**.** *Let $v \in V$ be an operator in a data flow program $D = (V, E)$. The* operator input data rate $R_{\text{in}}(v)$ *is the number of input records pre time unit of v. The* operator output data rate $R_{\text{out}}(v)$ *is the number of output records per time unit of v. Assuming a uniform data distribution over all tasks of an operator, the operator data rates can be computed based on the corresponding task data rates as follows:*

$$R_{\text{in}}(v) = dop(v) \cdot r_{\text{in}} \tag{3.12}$$

and

$$R_{\text{out}}(v) = dop(v) \cdot r_{\text{out}} \tag{3.13}$$

Definition 33 builds on the assumption of a uniform data distribution that is often applied in data-parallel systems. Introducing non-uniform data distributions into our model is interesting future work. If it is clear from the context to which node R_{in} and R_{out} belongs, we omit the parameter v in the formulas.

In the section above, we discussed the relationship between task and operator data rates. Next, we explain the relationship between the input and output data rate of a single task or operator. The output data rate of a task or operator is computed based on the input data rate and the selectivity.

Definition 34 (Operator and Task Output Data Rate)**.** *Given the input data rates r_{in} and R_{in} of a task and operator and the selectivity s. The corresponding output data rates are computed as follows:*

$$r_{\text{out}} = s \cdot r_{\text{in}} \tag{3.14}$$

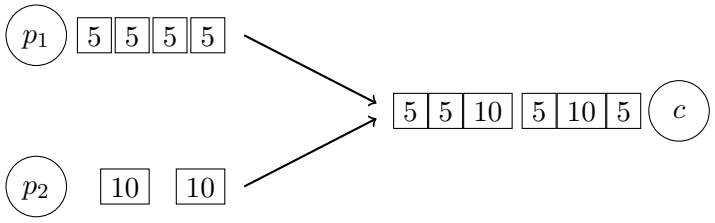

Figure 3.4: Data flow program with two producers (p_1 and p_2) configured with different output batch size and single consumer c.

and

$$R_{\text{out}} = s \cdot R_{\text{in}} \tag{3.15}$$

The above definitions are based on single operators. In the following, we discuss dependencies of consecutive operators. In particular the computation of input data rates and input batch sizes depend on the output data rates and output batch sizes of upstream producers.

Definition 35 (Operator Input Data Rate)**.** *Given a consumer $c \in V$ of a data flow program $D = (V, E)$ with corresponding producers $p \in V$. The consumer input data rate is the sum of the producer output data rates:*

$$R_{\text{in}}(c) = \sum_{\forall p \in V : \exists (p,c) \in E} R_{\text{out}}(p) \tag{3.16}$$

In the following, we describe the dependency of upstream output batch sizes and downstream input batch sizes of consecutive operators. In order to apply our cost function (Equation 3.8) that allows us to reason about the consumer performance, a single input batch size value is required. However, in general, a consumer can have multiple producers and all those producers could have a different output batch size. Hence, we introduce the notion of a consumer's *effective* input batch size that we compute as a weighted average of the output batch sizes of its producers.

Definition 36 (Effective Input Batch Size)**.** *The* effective input batch size *of a consumer c is the number of records over which the cost of a single dequeue operation is shared on average. The effective input batch size b_{in} is defined as:*

$$b_{\text{in}}(c) = \frac{\textit{input records per time unit}}{\textit{batches per time unit}} \tag{3.17}$$

In the remainder of this work we use the terms input batch size and *effective* input batch size interchangeably and denote both as b_{in}.

Example 5. *Assume two producers p_1 and p_2 with output batch size $b_{\text{out}}(p_1) = 5$ and $b_{\text{out}}(p_2) = 10$ and a single consumer c as shown in Figure 3.4. Furthermore, assume that both producers have an output data rate R_{out} of $1\,\text{s}^{-1}$. Thus, p_1 emits an output batch every 5 seconds while p_2 emits an output batch every 10 seconds.*

In this setup, c receives 3 batches in every 10 second interval—2 batches of size 5 and 1 batch of size 10. As discussed in Section 3.2.1, the cost of dequeuing an input batch is shared over all records within the batch. We observe that in our example

dequeuing a batch of size 5 *happens twice as often as dequeuing a batch of size* 10*. On average, c performs* 3 *dequeue operations to receive* 20 *records—*2 *dequeue operations providing* 5 *records each, and* 1 *dequeue operation that provides* 10 *records. Therefore, the cost of a single dequeue operation is shared over* $\frac{\#records}{\#batches} = \frac{20}{3} = 6.6$ *records on average in our example.*

Example 5 shows that the dequeuing cost of input batches is shared over a certain number of records on average. There are two parameters that determine this average number of input records: the different input batch sizes as well as the corresponding frequencies of arriving batches with different sizes. The frequency of arriving batches depends on the output batch size and the output data rate of the producer. We can generalize this to compute the effective input batch size b_{in} of a consumer as follows:[5]

Given a consumer c that receives input from n producers $p_1, \ldots, p_n$ with producer output data rates $R_{\text{out}}(p_i)$ and output batch sizes $b_{\text{out}}(p_i)$. We compute b_{in} of c by using Equation 3.17 of Definition 36 (we denote the input data rate in batches as R^b_{in}).

$$b_{\text{in}}(c) = \frac{R_{\text{in}}(c)}{R^b_{\text{in}}(c)} \tag{3.18}$$

The input data rate $R_{\text{in}}(c)$ is computed based on $R_{\text{out}}(p_i)$ using Equation 3.16. To compute the input data rate in batches R^b_{in}, we first compute the producer output data rate in batches. The number of output batches per time unit $R^{b_{\text{out}}}_{\text{out}}$ for a producer p_i is:

$$R^{b_{\text{out}}}_{\text{out}}(p_i) = \frac{R_{\text{out}}(p_i)}{b_{\text{out}}(p_i)} \tag{3.19}$$

Based on Equation 3.19 the input data rate in batches R^b_{in} of c is the sum over all output batches per time unit of the producers p_i:

$$\begin{aligned} R^b_{\text{in}}(c) &= \sum_{i=1}^{n} R^{b_{\text{out}}}_{\text{out}}(p_i) \\ &= \sum_{i=1}^{n} \frac{R_{\text{out}}(p_i)}{b_{\text{out}}(p_i)} \end{aligned} \tag{3.20}$$

Hence, the *effective input batch size* b_{in} of c is computed by utilizing Eq. 3.16 and Eq. 3.20 as substitutions within Eq. 3.18 as:

$$\begin{aligned} b_{\text{in}}(c) &= \frac{R_{\text{in}}(c)}{R^b_{\text{in}}(c)} \\ &= \frac{\sum_{i=1}^{n} R_{\text{out}}(p_i)}{\sum_{i=1}^{n} \frac{R_{\text{out}}(p_i)}{b_{\text{out}}(p_i)}} \end{aligned} \tag{3.21}$$

[5]We use operator data rates to compute the effective input batch size below. Using task data rates would be possible, too, and would yield the same result.

Equation 3.21 provides a way to compute the effective input batch size of a consumer solely based on properties of the corresponding producers: in particular the configured output batch sizes as well as the output data rates. This representation simplifies the application of our cost model in Chapter 4.

In this section, we focused on a task processing capacity. The input parameters of our model are single record processing cost, dequeue and enqueue cost, as well as the operator selectivity. Given the upstream producer output data rates and output batch sizes, we can compute the consumer processing capacity for any configured consumer parallelism and output batch size. In Chapter 4, we leverage our equations to optimize output batch sizes given different "performance goals". In the next section, we discuss network costs of our cost model.

3.3 Network Costs

Similar to processing costs, we model network costs based on the capacity concept as introduced in Definition 24. The network capacity is the maximum data rate in records per time unit that a task can receive or send. It depends on the underlying hardware as well as the record sizes. We model the hardware as maximum throughput of a network connection $\mathcal{N}$ in bits per second (called *bandwidth*). Furthermore, we assume that a network connection is exclusive for a single task and that input and output connections are independent. How $\mathcal{N}$ is determined is discussed in more detail in Section 4.2. In this section, we assume $\mathcal{N}$ to be given. The second parameter for the network capacity is the record size rs, i. e., the average size in bytes of a serialized record. Given $\mathcal{N}$ and the record size we define the network capacity as follows:

Definition 37 (Network Capacity)**.** *The* network capacity *is the maximum number of records that can be transferred between two consecutive tasks per time unit. Given $\mathcal{N}$ and rs, we compute the network capacity C_{N} as:*

$$C_{\mathrm{N}} = \frac{\mathcal{N}}{8 \cdot rs} \tag{3.22}$$

Since $\mathcal{N}$ is given in bits per second, we divide by a factor of 8 to translate from bits to bytes.

The network capacity C_{N} depends on $\mathcal{N}$ and the record size rs. In practice, not all records in a data stream have the exact same size. To incorporate different record sizes in our model, we use the *average* record size instead of individual record sizes. We denote average input and output record sizes of operators as rs_{in} and rs_{out}, respectively.[6] In the remainder of this work, we use the term *record size* to refer to the *average* record size for simplicity.

We point out that the network capacity models the maximum number of records that can be *transferred* over the network and is not to be confused with the *input network capacity* or the *output network capacity* of a task (Definition 28). In particular, the *output* network capacity models an *input* data rate. It does *not* describe

[6] We do not distinguish between operator and task input/output record sizes, because they are the same for an operator and its corresponding tasks.

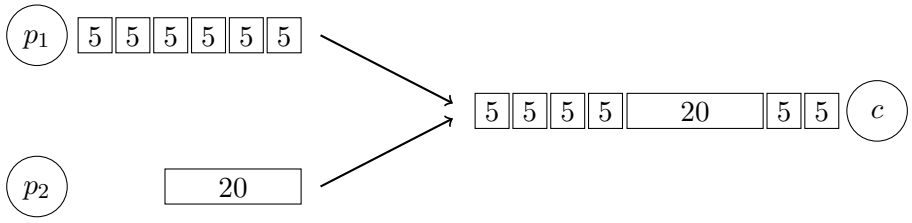

Figure 3.5: Data flow program with two producers (p_1 and p_2) with different output data rates and different output record sizes and a single consumer c.

the maximum output data rate that can be transferred to downstream tasks. We explain how the network capacity is used to define *input* and *output* network capacity in Section 3.3.1 and Section 3.3.2, respectively.

3.3.1 Input Network Capacity

The input network capacity C_i depends on $\mathcal{N}$ as well as on the input record size (Equation 3.22). Because consumers receive data from upstream producers, the producer output record sizes determine the consumer input record sizes. If a consumer has multiple upstream producers, it receives records of different sizes from different producers. Hence, we introduce an *effective* input record size that we compute as weighted average over all upstream output record sizes.[7]

Definition 38 (Effective Input Record Size)**.** *The* effective input record size *of a consumer c is the average record size in bytes over all input records. We define the effective input record size rs_in as:*

$$rs_\text{in}(c) = \frac{\textit{input bytes per time unit}}{\textit{input records per time unit}} \tag{3.23}$$

Example 6. *Assume two producers p_1 and p_2 with output record size $rs_\text{out}(p_1) = 5\,\text{B}$ and $rs_\text{out}(p_2) = 20\,\text{B}$ and a single consumer c as shown in Figure 3.5. Furthermore, assume that the producers have output data rates $R_\text{out}(p_1) = 6\,\text{s}^{-1}$ and $R_\text{out}(p_2) = 1\,\text{s}^{-1}$. Thus, p_1 emits $6\,\text{s}^{-1} \cdot 5\,\text{B} = 30\,\text{B/s}$ while p_2 emit $1\,\text{s}^{-1} \cdot 20\,\text{B} = 20\,\text{B/s}$. This implies that c receives $50\,\text{B/s}$ with a data rate of $7\,\text{s}^{-1}$ records. Within each second, c receives records with an average size of $\frac{\#bytes}{\#records} = \frac{50\,\text{B}}{7} = 7.1\,\text{B}$ in this example.*

As illustrated in Example 6, there are two parameters that determine the effective input record size of a consumer: the output record sizes as well as the corresponding output data rates of the producers. We generalize this to compute the effective input record size rs_in of a consumer as follows:[8]

[7] Computing the effective input record size as weighted average is the same approach that we use to compute the effective input batch size for an operator (c. f. Section 3.2.2).

[8] We use operator data rates to compute the effective input record size below. Using task data rates would be possible, too, and would yield the same result.

Given a consumer c that receives input from n producers $p_1, \ldots, p_n$ with producer data output rates $R_{\text{out}}(p_i)$ and output record sizes $rs_{\text{out}}(p_i)$. We compute rs_{in} of c by using Equation 3.23 from Definition 38 (we denote the input byte rate as R^{B}_{in}):

$$rs_{\text{in}}(c) = \frac{R^{\text{B}}_{\text{in}}(c)}{R_{\text{in}}(c)} \tag{3.24}$$

The input data rate $R_{\text{in}}(c)$ is computed using Equation 3.16. To compute the input byte rate $R^{\text{B}}_{\text{in}}(c)$ we first compute the producer output byte rates $R^{\text{B}}_{\text{out}}$. The output byte rate for each producer p_i is its record output rate times its output record size:

$$R^{\text{B}}_{\text{out}}(p_i) = R_{\text{out}}(p_i) \cdot rs_{\text{out}}(p_i) \tag{3.25}$$

Based on Equation 3.25 we compute the input byte rate R^{B}_{in} of c as sum over all the output bytes rates of producers p_i:

$$\begin{aligned} R^{\text{B}}_{\text{in}}(c) &= \sum_{i=1}^{n} R^{\text{B}}_{\text{out}}(p_i) \\ &= \sum_{i=1}^{n} \left(R_{\text{out}}(p_i) \cdot rs_{\text{out}}(p_i)\right) \end{aligned} \tag{3.26}$$

Hence, the effective input record size rs_{in} of c is computed by utilizing Eq. 3.16 and Eq. 3.26 together within Eq. 3.24 as:

$$\begin{aligned} rs_{\text{in}}(c) &= \frac{R^{\text{B}}_{\text{in}}(c)}{R_{\text{in}}(c)} \\ &= \frac{\sum_{i=1}^{n} \left(R_{\text{out}}(p_i) \cdot rs_{\text{out}}(p_i)\right)}{\sum_{i=1}^{n} R_{\text{out}}(p_i)} \end{aligned} \tag{3.27}$$

Given the effective input record size $rs_{\text{in}}(v)$ of an operator based on Equation 3.27, we compute the corresponding task input capacity C_{i} utilizing Equation 3.22 as:[9]

$$C_{\text{i}} = C^{rs_{\text{in}}}_{\text{N}} = \frac{\mathcal{N}}{8 \cdot rs_{\text{in}}(v)} \tag{3.28}$$

Hence, the input network capacity of a task (Equation 3.28) is directly derived from the network capacity (Equation 3.22). In the next section, we describe how the output network capacity of a task is computed.

[9] In the remainder of this work, we use the term *input record size* to refer to *effective* input record size for simplicity.

3.3.2 Output Network Capacity

To compute the output capacity of a task we cannot use Equation 3.22 directly, even if the average output record size is known. First, the output network capacity is defined with respect to the maximum *input* data rate of a task (Definition 28). Thus, it is required to map the output data rate that is transferred over the network to its corresponding input data rate. Second, the output data rate of a task is not the same as the output data *load*.

Definition 39 (Task Output Load)**.** *The* output load r_{load} *of a task is the number of records per time unit that must be transferred over the network to downstream consumer tasks.*

Similarly to operator capacity, input and output data rate, the *operator output load* is defined as:

Definition 40 (Operator Output Load)**.** *The* output load R_{load} *of an operator is the number of records per time unit that must be transferred over the network to downstream consumers. Given an operator v with $dop(v)$ and tasks with output load r_{load}. The operator output load is:*

$$R_{\text{load}}(v) = dop(v) \cdot r_{\text{load}} \tag{3.29}$$

Operators may have multiple downstream consumers that all receive a full copy of the output stream. For this case, each record must be transferred over the network multiple times, resulting in increased network load. We use the term *fan-out* in our model [CcR+03], that is the number of downstream consumer of an operator, to compute the output load based on the output rate.

Definition 40 (Task Output Load (cont.))**.** *Given a task t with output data rate r_{out} and corresponding operator fan-out $f(v)$. The task output load r_{load} is computed as:*

$$r_{\text{load}}(t) = f(v) \cdot r_{\text{out}}(t) \tag{3.30}$$

To compute the maximum possible output data rate of a task t, it is required to consider the task output load and the downstream network capacity based on the corresponding operator output record size $rs_{\text{out}}(v)$ (Definition 37):

$$\max(r_{\text{load}}(t)) = C_{\text{N}}^{rs_{\text{out}}} = \frac{\mathcal{N}}{8 \cdot rs_{\text{out}}(v)} \tag{3.31}$$

Combining Equation 3.30 and Equation 3.31 yields:

$$\begin{aligned} \max(r_{\text{out}}(t) | r_{\text{load}}(t) \leq C_{\text{N}}^{rs_{\text{out}}}) &= \frac{\max(r_{\text{load}}(t))}{f(v)} \\ &= \frac{\mathcal{N}}{8 \cdot rs_{\text{out}}(v) \cdot f(v)} \end{aligned} \tag{3.32}$$

The maximum possible output data rate of a task t can be mapped to the output network capacity C_{o} (i. e., the maximum input data rate such that the task load is

smaller than the downstream network capacity), using the corresponding operator selectivity $s(v)$:

$$\begin{aligned} C_{\text{o}} &= \max(r_{\text{in}} | r_{\text{load}} \leq C_{\text{N}}^{rs_{\text{out}}}) \\ &= \frac{\max(r_{\text{out}} | r_{\text{load}} \leq C_{\text{N}}^{rs_{\text{out}}})}{s(v)} \\ &= \frac{\mathcal{N}}{8 \cdot rs_{\text{out}}(v) \cdot f(v) \cdot s(v)} \end{aligned} \tag{3.33}$$

A task's output network capacity depends on the operator output record size, selectivity, and fan-out. A small selectivity or small fan-out increases the capacity, while a large selectivity or a large fan-out decreases the capacity. Example 7 illustrates this relationship.

Example 7. *Assume a filter task with input data rate* $r_{\text{in}} = 1000\,\text{s}^{-1}$ *and operator selectivity* $s = 0.5$*, fan-out* $f = 4$*, and an input record size* rs_{in}*. Because the operator is a filter we derive that the output record size is the same as the input record size, i. e.,* $rs_{\text{out}} = rs_{\text{in}}$*. Hence, we can transfer the same total amount of records on the input as well as the output network connection. Since we have a fan-out of* 4*, the task has to send* 4 *result records downstream (i. e., over the network) for each input record. Last, for each input record there are* 0.5 *output records on average and the output data rate is* $r_{\text{out}} = s \cdot r_{\text{in}} = 500\,\text{s}^{-1}$*.*

Let's assume further that the network capacity is $1000\,\text{s}^{-1}$ *that is equal to the input data rate. Even if neither input data rate nor the output data rate is larger than the network capacity, the output load is larger, i. e.,* $r_{\text{load}} = f \cdot r_{\text{out}} = 4 \cdot 500\,\text{s}^{-1} = 2000\,\text{s}^{-1}$*. This implies that the task cannot process its input data rate.*

Overall, the output *network capacity of the filter task is* $C_{\text{o}} = \frac{1000\,\text{s}^{-1}}{f \cdot s} = \frac{1000\,\text{s}^{-1}}{4 \cdot 0.5} = 500\,\text{s}^{-1}$ input *records per second, which is smaller than the input data rate* $r_{\text{in}} = 1000\,\text{s}^{-1}$*.*

We model the output network capacity as maximum number of *input* records per time unit instead of maximum load. Using maximum load might seem more intuitive, however, our approach simplifies the overall model and optimization algorithm in Chapter 4, because processing capacity and network input capacity are defined as input records per time unit. Defining the output network capacity based on input records per time unit (instead of maximum load), allows us to combine all three capacities into an overall operator capacity in a straightforward manner. Modeling the output network capacity as maximum load would prohibit (or complicate) Definition 28.

The above definitions of network costs complete our cost model. Table 3.1 summarizes all introduced input and cost model parameters. We discuss important design aspects of a batching implementation for queues in distributed, data-parallel systems in the next section.

3.4 Batching Layer

To design a record batching layer for distributed, data-parallel streaming systems, we need to consider the system model as discussed in Section 2.2 and Section 2.3. For

Table 3.1: Cost Model Parameters

Parameter	Abbr.	Notes
data flow program	$D = (V, E)$	nodes V and edges E
execution graph	$EG = (T, F)$	tasks T and connections F
configuration	Γ	*c. f. Definition 41 (Chapter 4)*
degree of parallelism	*dop*	
input/output batch size	b_{in} / b_{out}	in number of records
selectivity	s	can be larger than 1
fan-out	f	number of consumers for a producer
input/output record size	rs_{in} / rs_{out}	in bytes (B)
operator input/output rate	R_{in} / R_{out}	in records per second (s^{-1})
task input/output rate	r_{in} / r_{out}	in records per second (s^{-1})
task/operator output load	r_{load} / R_{load}	in record per second (s^{-1})
workload	$\mathbb{W}$	*c. f. Definition 42 (Chapter 4)*
data flow capacity	$\mathbb{C}$	
set of data flow capacities	$\mathbb{C}^{\dagger}$	
operator capacity	$C(v)$, $v \in V$	in records per second (s^{-1})
task capacity	$C(t)$, $t \in T$	in records per second (s^{-1})
processing capacity	C_{p}	in records per second (s^{-1})
input capacity	C_{i}	in records per second (s^{-1})
output capacity	C_{o}	in records per second (s^{-1})
processing cost (per record)	c_{p}	in seconds (s)
dequeuing cost (per record or batch)	c_{fetch}	in seconds (s)
cpu cost (per record)	c_{cpu}	in seconds (s)
emitting cost (per record or batch)	c_{emit}	in seconds (s)
network capacity	C_{N}	in records per second (s^{-1})
network bandwidth	$\mathcal{N}$	e. g., 1 Gbit/s

example, an operator instance may have multiple downstream consumers, each with a different degree of parallelism. Hence, an operator must partition its output records into distinct batches based on the partitioning scheme and consumer parallelism, to ensure correct data repartitioning. In this section, we design a batching layer for data parallel systems. We describe how a batching layer may be designed in this

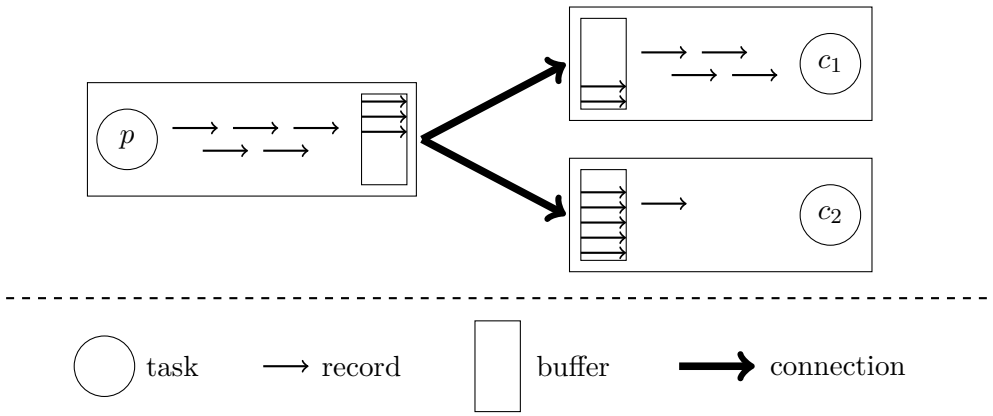

Figure 3.6: Producer task p with single output buffer and two consumer tasks (c_1 and c_2) connected via random or broadcast connection pattern.

section in general. Additionally, we provide some details of our implementation using the open source system Apache Storm [ASFf, TTS+14] in Section 4.3.

In the following, we distinguish between a logical input stream and its physical substreams, i. e., partitions (Section 2.2.1 and Section 2.3.2). Substreams are processed by independent tasks and data is redistributed between all tasks of a producer-consumer pair. To ensure correct data distribution, batching must consider the four common partitioning patterns as discussed in Section 2.2.2. Depending on the partitioning pattern a different batching scheme is applied. Below, we present three different batching schemes that use a different number of buffers to assemble record batches. The number of buffers impacts the code complexity, memory requirement, as well as processing latency for each batching scheme.

Batching for Random Partitioning

Random or broadcast distribution do not require special data partitioning. Hence, we use a single output buffer to assemble record batches. Each time the output buffer is full, i. e., a batch is completed, the system either sends the batch of records to one (random) or to all (broadcast) consumer tasks. Figure 3.6 shows one producer with one task p connected to one consumer with two tasks c_1 and c_2. The producer sends data to the consumer via two substreams depicted as bold connection arrows. The producer inserts output records (depicted as small arrows) into the output buffer (depicted as rectangle). We assume a batch (i. e., buffer) size of six records in our example. Each time the output buffer is full, the producer sends a batch with six records to one consumer task, assuming random partitioning. For broadcasting, the producer sends each batch to all consumer tasks. The consumer tasks receive the record batches and process each record within a batch individually.

Using a single buffer for batching implies (1) low code complexity, (2) low memory requirements, and (3) low latency. (1) Because there is only a single buffer, each record is appended to the same buffer and data partitioning is no concern. (2) The producer buffers at most b_{out} records (independent of the number of consumer tasks), i. e., the memory requirement is minimal for assembling a batch of b_{out} records. (3) A record batch is completed after b_{out} output records are emitted by p without any

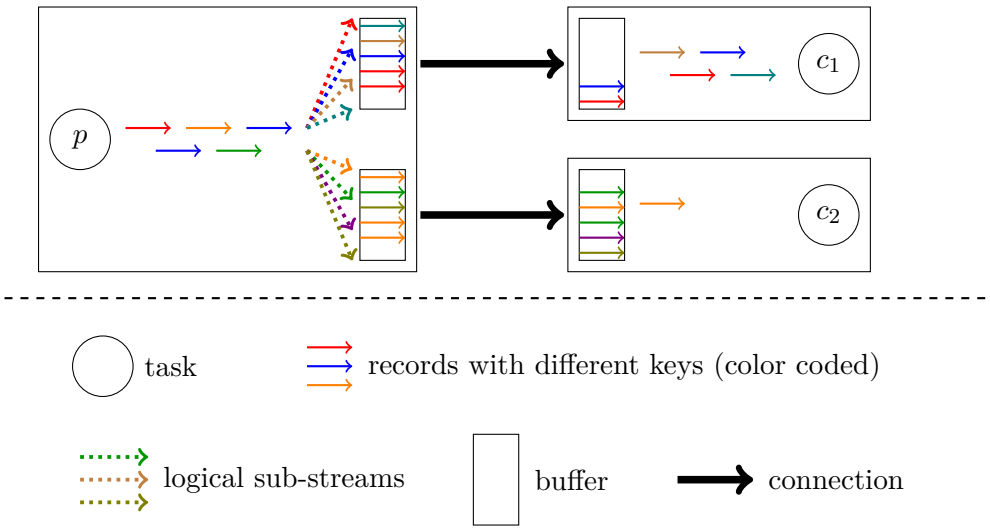

Figure 3.7: Producer task p with two output buffers and two consumer tasks (c_1 and c_2) connected via hash- or range-partitioning connection pattern.

additional delay, i. e., latency is minimal as a batch cannot be sent downstream before b_{out} output records are available.

A single output buffer for batching provides low code complexity, memory requirements, and latency. However, it can only be used for random partitioning or broadcasting. For hash-based or range-based partitioning, a single output buffer is not sufficient but would result in incorrect repartitioning. Data of different partitions that must be processed by different tasks, would be contained in a single batch that is sent to a single tasks. Therefore, it is necessary to use multiple distinct output buffers to separate data of different partitions from each other, as discussed in the next section.

Batching for Key-Based Data Distribution

In Section 2.3.2 we introduced two key-based data partitioning techniques: hash-based and range-based partitioning. Both, hash-partitioning and range-partitioning are based on some grouping attributes within the records. We call those grouping attributes *keys*[10]. All records with the same key are grouped together and form a logical substream, and logical substreams are grouped into physical substreams. To guarantee correct data partitioning, a batch can only contain records that must be transferred via one physical substream to one consumer task. To explain the underlying concept of key-based batching, we first describe a simplified batching technique that only works for producers with a single consumer. Key-based batching for the general case with multiple consumers is discussed later.

Figure 3.7 shows one producer with one task p connected to one consumer with two tasks c_1 and c_2. The producer sends data to the consumer via two substreams depicted as bold connection arrows. We assume a batch size of six records and eight different record keys (indicated by different colors). Task c_1 processes records with red, blue, brown, and teal keys and task c_2 processes records with orange, green,

[10] A grouping key is not a unique identifier (or primary key) like in the relational data model.

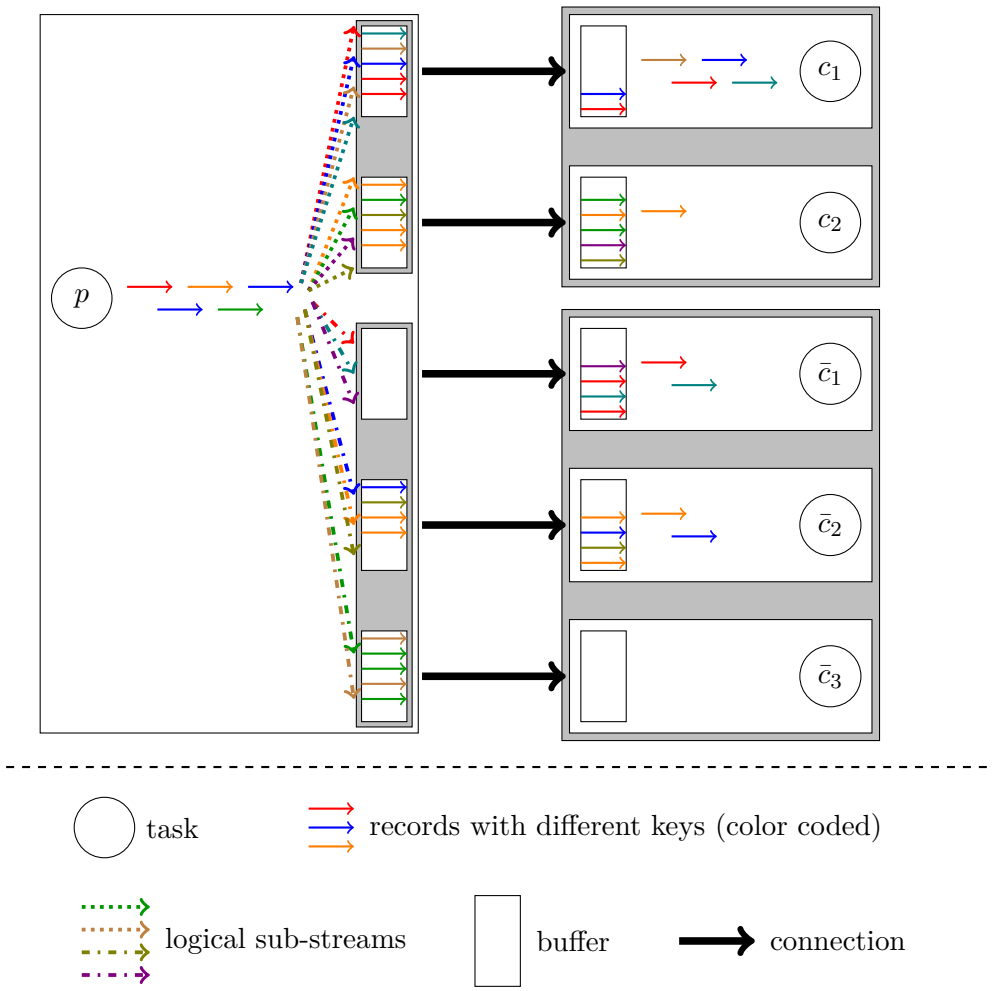

Figure 3.8: Producer task p with distinct output buffers and two consumers with different degree of parallelism, connected via hash- or range-partitioning connection pattern.

violet, and olive keys. Within the producer, output records (depicted as solid colored arrows) are inserted into the output buffers (depicted as rectangles) depending on the record key. While there are eight logical substreams (indicated by the dotted colored arrows), the producer splits its output into two physical substreams because there are only two consumer tasks. For each physical substream, one buffer is used to assemble a batch of records and each full batch is sent to exactly one consumer task. The consumer tasks receive the record batches, and process each record within a batch individually. Because one output buffer is used for each task, and there is a one-to-one mapping between output buffers and consumer tasks, records are correctly redistributed based on the partitioning function.

The above described batching technique is simplified and does not work if there are multiple downstream consumers with different degree of parallelism or different partitioning functions. In the following we describe a generic batching technique that work for all cases, namely *distinct batching scheme* and *shared batching scheme*. The advantage of distinct batches is that it uses a smaller number of output buffers

compared to the shared batching scheme. A reduced number of buffers implies a reduced memory requirement as well as reduced latency compared to the shared batching scheme. We additionally introduce the shared batching scheme because it can be implemented in user space, which may be a desired property as discussed in more detail below.

Distinct Batching If a producer has multiple consumers, each consumer may have a different degree of parallelism and may use a different partitioning strategy. To ensure that each batch only contains records for a single consumer task, it is possible to maintain one batching buffer per task over all consumers. Buffers are logically grouped into buffer pools—one pool per consumer—each containing one buffer per associated consumer task. Each output record must be transferred once to each consumer, and hence, it is inserted into one buffer per buffer pool. For each buffer pool, hash or range partitioning is used considering the degree of parallelism of the corresponding consumer. Hence, each record is inserted into one buffer per pool, independently to all other buffer pools. The initial example from Figure 3.7 is a special case for a single consumer. It can be applied to multiple consumers only if all consumers use the same partitioning strategy, based on the same key, and have the same degree of parallelism. For this case, all buffer pools are exact copies of each other, and therefore it is possible to "merge" them.

The general case is depicted in Figure 3.8. The figure illustrates a single producer task p with two logical consumers. Similar to the example in Figure 3.7 we assume a batch size of six records and eight different record keys (indicated by different colors). The first logical consumer is the exact same as in Figure 3.7. Task c_1 processes records with red, blue, brown, and teal keys and task c_2 processes records with orange, green, violet, and olive keys. Additionally, there is a second consumer with three tasks $\bar{c}_1$, $\bar{c}_2$, and $\bar{c}_3$. In our example, both consumers use the same grouping strategy, however, the second consumer has three tasks, and thus, the eight logical substream are assigned to the three tasks differently compared to the first consumer. Task $\bar{c}_1$ processes records with red, teal, and violet keys, task $\bar{c}_2$ processes records with blue, orange, and olive keys, and task $\bar{c}_3$ processes records with green and brown keys. Since there are two logical consumers, the producer uses two buffer polls (depicted as gray rectangles); one for each consumer. Within the producer, output records (depicted as solid colored arrows) are inserted into one output buffer (depicted as rectangles) per buffer pool (indicated by the dotted or dashed arrows—each key color is used once per buffer pool) depending on the record key. Hence, the eight logical substreams are grouped into different physical substreams per consumer. For each physical substream, one buffer is used to assemble a batch of records and each full batch is sent to exactly one consumer task. The consumer tasks receive the record batches, and process each record within a batch individually. Since one output buffer is used for each task, and there is a one-to-one mapping between output buffers and consumer tasks, records are correctly redistributed based on the partitioning function.

Using distinct buffers for each consumer task implies increased (1) code complexity, (2) memory requirement, and (3) latency compared to using a single buffer as discussed above. (1) There are multiple buffers and multiple buffer pools, and each record must be inserted using a different partitioning function per buffer pool.

(2) The required overall buffer size is $(\sum_C dop(C)) \cdot b_{\text{out}}$ records, i. e., the required memory grows with larger consumer parallelism. (3) Assuming a uniform data distribution, a record batch is completed on average after $\overline{dop} \cdot b_{\text{out}}$ output records are emitted by p.[11] Hence, records are buffered longer until a batch is completed compared to using a single buffer, and thus, latency increases.

Distinct batching requires to send each output batch to exactly one task of exactly one consumer. This is easy to implement within a stream processing system. However, there are cases for which the stream processing system does not implement batching and users would like to add this capability on top of the system. For this case, the user code does not forward single records but batches of records transparently to the processing system. Because the stream processing system is not aware of this change, it forwards each batch to one task of each downstream consumers. Therefore, the distinct batching scheme cannot be applied because it requires that a batch is sent to one task of one specific consumer only. To address this issue, we introduce the shared batching scheme as discussed in the next paragraph.

Shared Batching The shared batching scheme is a technique that allows for a fully transparent batching layer implementation on top of a stream processing system without the need to alter the underlying system. If the underlying system does not support batching and cannot be modified (i. e., not open-source), shared batches can be used to increase system throughput.

For this case, the stream processing system is not aware that it transfers batches of records and it sends each batch to one task of each logical consumers. Therefore, it is required to assign records to batches such that the system can send each batch to all logical consumers without violating the original partitioning strategy. To achieve this, shared batches uses a polynomial number of output buffers based on the number of tasks per consumer. Given a producer P with consumers $C_1, \cdots, C_n$, the number of used buffers is:

$$\prod_{C \in \{C_1, \cdots, C_n\}} dop(C)$$

Using our example from Figure 3.8, the consumers A and B have a *dop* of 2 and 3, respectively. Shared batches uses $2 \cdot 3 = 6$ buffers that are conceptually represented as a matrix with 2 rows and 3 columns (Figure 3.9). The matrix entries represent all possible consumer task combinations from different consumers that may share records within one batch. For each record, a buffer is selected a follows: The key attributes of consumer A are used to select the row (based on the consumer partitioning function pf_A) and the key attributes of consumer B are used to select the column (base on the consumer partitioning function pf_B). The record is inserted in the buffer corresponding to the computed position in the matrix. Figure 3.9 depicts which key is inserted into what buffer using color coding. For example, the top-left buffer, contains records with red or teal key only. In our example, a record with blue key is inserted into the buffer of tasks c_1 and $\bar{c}_2$. This buffer selection scheme ensures that records that are added to a batch obey the partitioning requirements of both consumers. Therefore, each buffer can be sent to each consumer without

[11] We denote the *average* degree of parallelism of all consumers of a producer as $\overline{dop}$.

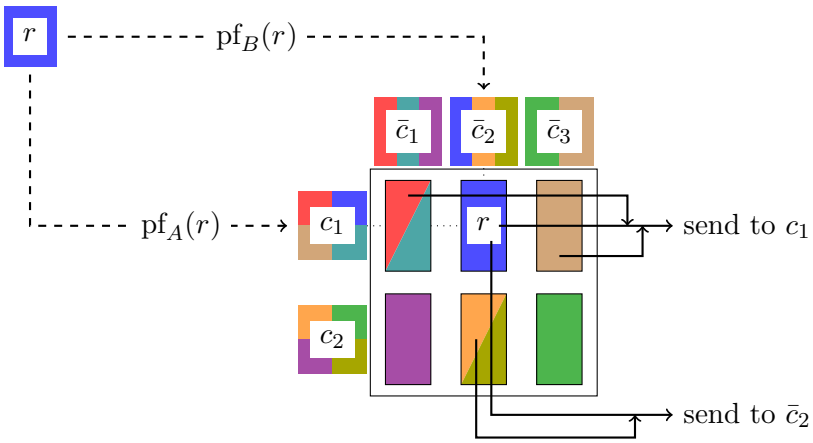

Figure 3.9: Matrix of 6 buffers for two logical consumers A and B with $dop(A) = 2$ and $dop(B) = 3$.

violating the partitioning. While our example uses two consumers, and thus, a matrix (i. e., a 2-dimensional array), the same pattern extends to n consumers using an n-dimensional array.

Figure 3.10 depicts the same producer and consumer tasks as used in Figure 3.8 and both consumers use the same partitioning. However, the producer uses shared batching. We point out that there are no buffer pools and each record is inserted into exactly one buffer.[12] Furthermore, each buffer is transferred twice—once for each consumer. All batches are assembled to not violate the partitioning. In our example with only 8 different keys, only two buffers have two different keys: The first buffer contains records with red and teal keys, and the fifth buffer contains records with orange and olive keys. Notice that red and teal keys are processed by a single task of each consumer, namely c_1 and $\bar{c}_1$. Hence, it is correct to add red and teal keys to the same batch. Similarly, records with orange and olive keys are processed together in both consumers. As a counter example, it would be incorrect to add records with red and blue key to the same buffer: while red and blue keys are processed by a single task of the first consumer (namely c_1), records with red and blue keys are processed by different tasks of the second consumer (namely $\bar{c}_1$ for red and $\bar{c}_2$ for blue). Hence, sending a batch containing records with red *and* blue keys, to either $\bar{c}_1$ or $\bar{c}_2$ would be incorrect.

Shared batches is an over-partitioning strategy: Records that are processed by one consumer task are added to different batches and each consumer task receives batches from multiple buffers. For example c_1 processes all records with red, blue, brown, and teal keys (c. f. Figure 3.8). In the shared batching scheme, records with red and teal keys are added to batch number one, records with blue keys are added to batch number two, and records with brown keys are added to batch number

[12] Conceptually, the buffers are arranged in a matrix as depicted in Figure 3.9. Figure 3.10 depicts the six buffers without background color coding and arranged below of each other for ease of presentation. The color coded dotted and dashed logical substreams indicate the same partition as in Figure 3.9 though: The top three buffers correspond to the top row, and the bottom three buffers correspond to the bottom row.

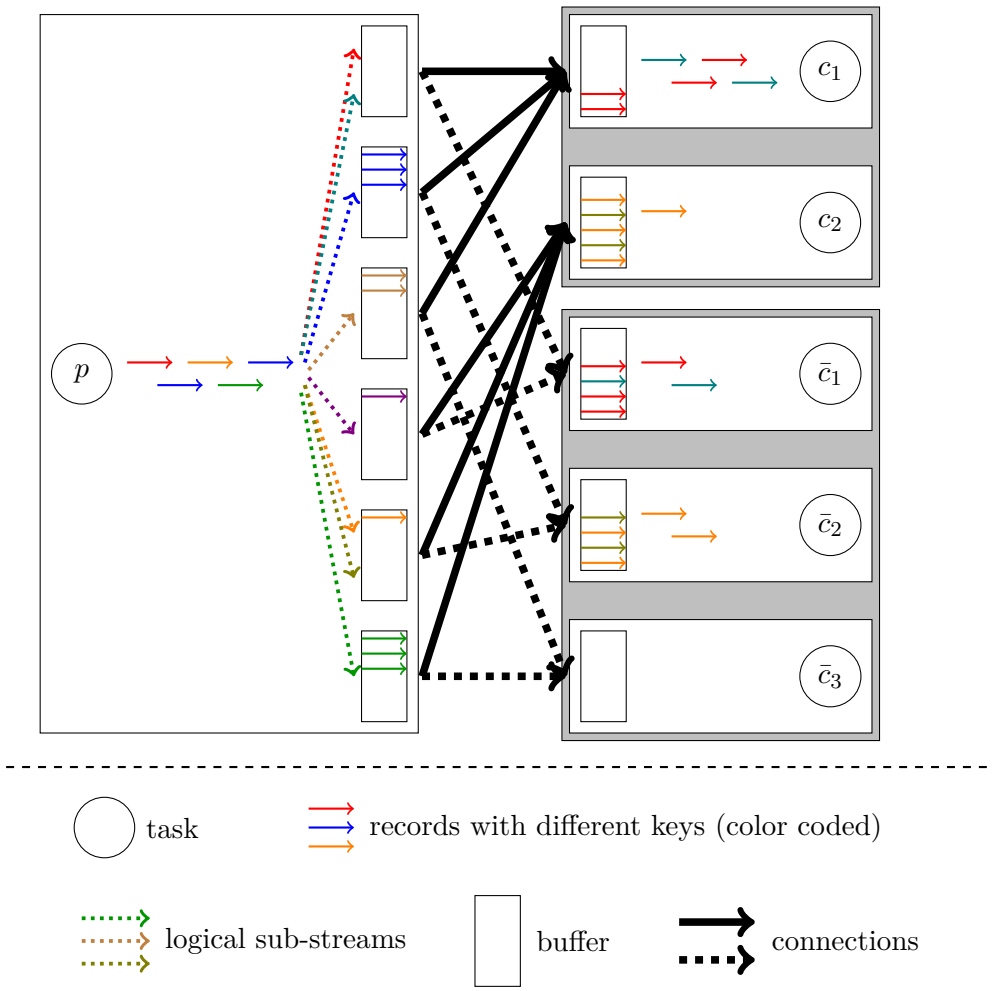

Figure 3.10: Producer task p with shared output buffers and two consumers with different degree of parallelism, connected via hash- or range-partitioning connection pattern.

three. Therefore, it is required to send all those batches to task c_1 as depicted in Figure 3.10.

We described shared batches scheme for two logical consumers. However, shared batches work the same way for n consumers by arranging all buffers logically in an n-dimensional array. The disadvantage of shared batches compared to distinct batches is the use of more output buffers. Hence, there is an increased main memory consumption within the batching layer and an increased latency because buffers do not fill up as quickly if there are more buffers. Another impact of shared batches is that record offset order (Section 2.4.4) is not preserved. The relative offset order is only preserved for records with the same key.

3.5 Related Work

Throughput, capacity, and output data rate are define similarly to our work in Aurora [CcC+02]. However, Aurora uses its cost model for operator scheduling as

it is a centralized and single threaded system. There is also no notion of network cost or batching (their *train scheduling* approach is purely related to scheduling). Borealis [XZH05] uses the same cost model as Aurora for operator placement. The load of a single server is the accumulated cost over all operators executed by the server. Borealis aims to assign operators dynamically to ensure load balancing within the processing cluster. While our cost model shares a common basis with Aurora/ Borealis, we aim to estimate the execution cost of parallel data flows, to reduce the execution cost via batching, and to optimized operator parallelism.

Daum et al. [DLB+11] present a method for cost model calibration. Similar to our model, the system and operators are treated as black-boxed. To determine cost model parameters, multiple queries with different characteristics are executed while resource consumption is measured. A similar approach may be applicable to calibrate our cost model.

There are many rate-based cost models for data stream processing. Those models usually assume to know the semantics or even implementation of the modeled operators. Viglas et al. [VN02] propose one of the first rate-based cost models, considering selections (i. e., filter), projections, and joins (so-called SPJ queries). The goal of their optimization is to maximize the *output* data rate. Ayad and Naugthon [AN04] use their cost model to rewrite query plans with the goal to minimize resource consumption. If the system is overloaded, they propose to optimize for a maximized output data rate similar to Viglas et al. Many papers focus on the optimization of multi-way windowed stream joins [KNV03, GO03b, GC06, CKSV08, HAE08] Those model compare different join algorithms, and try to find an optimal operator tree. They usually considering window sizes, query execution interval, and time-granularity. While all those cost models are rate-based, the optimization goal is orthogonal to ours. However, it seems possible to combine those model with ours and to model the actual processing costs of an operator not as a black-box but with operator specific knowledge.

Improving the throughput of distributed, data-parallel stream processing system via batching is a known technique [LWK12, DZSS14]. Existing work does not model the impact of batching but instead focuses on dynamic batch size adjustments base on observed system load. We consider our cost-model-based and predictive approach complementary to those reactive approaches.

3.6 Summary

In this chapter, we introduced a rate-based cost model for streaming data flow programs, considering CPU and network costs. Our model treats operators as black-boxes while considering data parallel operator execution as well as record batching. We base our model on the concept of operator/task and data flow capacity, and describe inter and intra operator dependencies. A detailed model like ours, provides insight in the performance of streaming data flow programs and helps to understand operator dependencies.

While rate-based cost models are common in data stream processing, they are usually used for query optimization similar to optimizers in relational database systems. In contrast, the goal of our model is to describe the performance of immutable data flow programs with black-box operators. We use our model in Chapter 4 to

optimize the data flow configuration (formally defined later) to avoid bottlenecks and over provisioning. We believe that adaptive and reactive optimization methods are not sufficient [FAG+17] and propose to enhance those method with a cost model like ours to improve dynamic scaling.

Additionally, we provided a detailed description on the design of our batching layer for data-parallel stream processing systems and discussed different trade-offs and optimization opportunities when employing record batching.

Chapter 4

Data Flow Optimization

Contents

In Chapter 3, we presented our cost model for streaming data flow programs. Based on this cost model, we introduce algorithms for bottleneck detection and data flow optimization in this chapter. In particular, we present an algorithm in Section 4.1 that takes a data flow program, its configuration, and workload as input, and outputs bottlenecks in the data flow program.[1] Additionally, it computes the effective throughput for each operator. The second algorithm (Section 4.2) focuses on performance optimization. The algorithm takes a data flow program and a workload as input and compute a configuration based on an optimization criteria. We define the corresponding optimization problem formally and prove that our algorithm computes an optimal solution for it.

As described in Chapter 2, a data flow program is translated into an execution graph (Definition 4) using a function *dop* that specifies the degree of parallelism for each operator. Furthermore, each operator may be executed with different batch sizes (Section 3.2.1). We assume that the output batch size for each operator is specified similar to the degree of parallelism. As explained in Corollary 1, input batch sizes cannot be specified, but are computed based on the output batch sizes

[1] *Configuration*, *workload*, and *bottlenecks* are defined below.

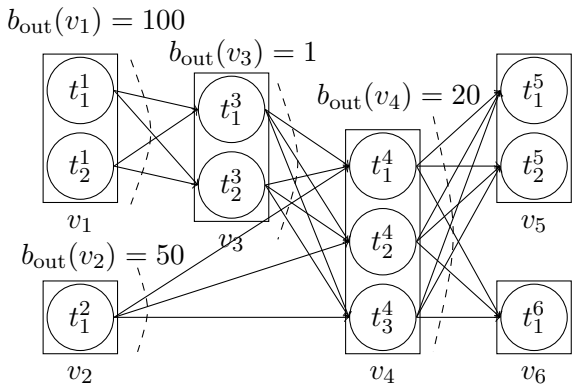

Figure 4.1: Execution graph with parallelism and output batch sizes from Example 8.

of upstream producers. We unify both parameters as a *configuration* of a data flow program, defined as follows:

Definition 41 (Configuration)**.** *Let $D = (V, E)$ be a data flow program. A* configuration *of D is a function $\Gamma : V \to \mathbb{N} \times (\mathbb{N} \cup \{\perp\})$*

$$\Gamma(v) = \langle \mathit{dop}(v), b_{\text{out}}(v) \rangle \tag{4.1}$$

where $v \in V$ is an operator in the data flow with $\mathit{dop}(v)$ and $b_{\text{out}}(v)$ being the degree of parallelism and the output batch size of v. For sinks, no output batch size is assigned, indicated via $\perp$.

Example 8. *Consider the data flow program from Example 1 and a configuration Γ for D as shown in the following table:*

	v_1	v_2	v_3	v_4	v_5	v_6
dop	*2*	*1*	*2*	*3*	*2*	*1*
b_{out}	*100*	*50*	*1*	*20*	$\perp$	$\perp$

The top row shows the operators and each column contains the dop and the output batch size of one operator. Operators v_5 and v_6 do not have any batch size configured (denoted $\perp$) since they are both sinks and do not have outgoing edges. The resulting execution graph with its batch size configuration is depicted in Figure 4.1.

Additionally to a configuration, a user may specify a *workload* (as for the algorithms presented in Section 4.1 and Section 4.2) that is defined as follows:

Definition 42 (Workload)**.** *A* workload *for a data flow program is the expected input data rate per source. Given a data flow program D with sources $S = \langle s_1, \ldots, s_n \rangle$ a workload $\mathbb{W}$ for D is:*

$$\mathbb{W}(D) = \langle I_1, \ldots, I_n \rangle \tag{4.2}$$

where I_i denotes the expected number of input records per time unit for s_i.

A workload is defined as a vector, similar to the data flow capacity (Definition 25). Since both are represented as vectors, it is possible to compare workloads to capacities in the following sections to each other.

4.1 Bottleneck Detection and Throughput Prediction

In this section, we present an algorithm that takes a data flow program, its configuration, and a workload as input and identifies *bottlenecks* in the execution graph. Additionally, the algorithm computes the throughput of each operator in the data flow program. Before we present the algorithm, we formally define *bottlenecks* as follows:

Definition 43 (Bottleneck)**.** *An operator v is a* bottleneck *in a data flow program if its input data rate R_{in} is larger than its capacity C (Definition 27):*

$$v \text{ is a bottleneck} \iff R_{\text{in}}(v) > C(v) \tag{4.3}$$

For this case, we say that the operator is overloaded*. We further distinguish between* CPU bottlenecks *and* network bottlenecks*, and say an operator is* CPU bound *or* network bound*, respectively. An operator is CPU bound if its input data rate is larger than its processing capacity C_{p}:*

$$v \text{ is CPU bound} \iff R_{\text{in}}(v) > C_{\text{p}}(v) \tag{4.4}$$

An operator is network bound, if its input data rate is larger than its input (C_{i}) or output (C_{o}) capacity (c. f. Section 3.3.2), respectively:

$$\begin{aligned} v \text{ is input network bound} &\iff R_{\text{in}}(v) > C_{\text{i}}(v) \\ v \text{ is output network bound} &\iff R_{\text{in}}(v) > C_{\text{o}}(v) \end{aligned} \tag{4.5}$$

If there exists either a CPU bottleneck or a network bottleneck in a data flow program, we say that the data flow program is *CPU bound* or *network bound*. The above definition only applies for a data flow program given its configuration. Without a configuration the capacity of a data flow program is not defined.

In Chapter 3, we described how the output data rate of an operator is computed (Equation 3.13). This definition does not take into account that an operator might be overloaded. Thus, we re-define the computation of the operator output data rate as follows:

$$R_{\text{out}} = s \cdot \min\{R_{\text{in}}, C\} \tag{4.6}$$

If an operator is a bottleneck, its output data rate is not determined by its input data rate, but by its capacity (c. f. Equation 4.3). Based on our definition of bottlenecks, we describe our *bottleneck detection* algorithm in the next section (Section 4.1.1). In section Section 4.1.2, we compute the *effective throughput* for each operator (and thus, the data flow throughput) for the case that a bottleneck is detected.

4.1.1 Bottleneck Detection

In this section, we introduce an algorithm that identifies all operators in a data flow program that are overloaded, taking a data flow program, its configuration, and workload as input. This algorithm may be applied if a user wants to verify a manually specified configuration. We first explain the high level idea of our bottleneck detection algorithm and present it formally later.

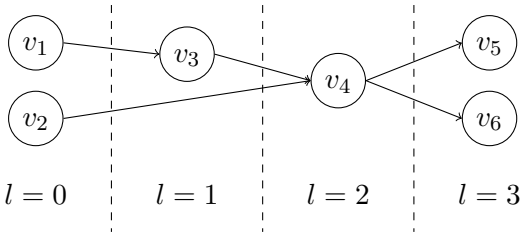

Figure 4.2: Operator levels of the data flow program from Example 1.

Given a workload and configuration of a data flow program, we identify bottlenecks in the corresponding execution graph as follows: for each operator in the data flow program, we compute its input data rate and capacity. We then compare the input data rate with the operator capacity. If the input data rate exceeds the capacity, the operator is overloaded, and thus, it is a bottleneck (Definition 43). The input data rate of sources is known from the provided workload. For all other operators, the input data rate depends on the output data rates of their upstream producers. We exploit this producer-consumer dependency in our bottleneck detection algorithm: the algorithm starts at the sources and traverse the whole data flow graph step by step until in reaches the sinks. To formally describe our algorithm, we introduce *levels* of a data flow graph as follows:

Definition 44 (Levels of a Data Flow Graph)**.** *For each operator in a data flow graph $D = (V, E)$, we assign a* (operator) level*, a natural number describing "how far" an operator is away from a source. We define levels $l \in \mathbb{N}_0$ recursively for all $v \in V$ starting at the sources that are assigned level* 0 *as follows:*

$$l(v) = \begin{cases} 0 & \text{if } \nexists v' : (v', v) \in E \\ \widehat{l}(v) + 1 & \text{otherwise} \end{cases} \tag{4.7}$$

with

$$\widehat{l}(v) = \max\{l(v') | \forall v' : (v', v) \in E\}$$

Example 9. *Let $D = (V, E)$ be the data flow graph from Example 1. Figure 4.2 depicts D including the levels for all operators. The two sources v_1 and v_2 both have level* 0*. Operator v_3 has level* 1*, because its only predecessor v_1 has level* 0*. For v_4, we need to consider both predecessors and the maximum of their levels. Therefore, $l(v_4) = 2$ because $\widehat{l}(v_4) = \max\{l(v_2), l(v_3)\} = \max\{0, 1\} = 1$. Finally, both v_5 and v_6 have level* 3*.*

Our traversal strategy is different to a breath-first search (BFS) traversal. For example, in BFS operators v_3 and v_4 would both be considered in the second recursive step of the algorithm, while in our case only v_3 is considered. The difference is that recursive levels in BFS are defined as the *minimum* distance to sources (i. e., the "root" node), while we define levels as the *maximum* distance.

Algorithm 1 shows our bottleneck detection algorithm. After the initialization phase (Lines 7-8) the algorithm traverses the complete data flow program (Lines 10-28). In each round, it processes all operators of one level (Line 11) starting with the sources at level 0 (Line 8). Processing a single operator consists of four steps. (1)

Algorithm 1: Bottleneck Detection

```
1  Input: data flow D = (V, E); configuration Γ(D); workload 𝕎(D)
2  Output: all operators B that are overloaded
3
4  Def: V(l) ← {v ∈ V | l(v) = l}                    // all operators of level l
5  S ← V(0)                                                    // get sources
6
7  B ← {}                                              // all found bottlenecks
8  l ← 0                                                    // start at level 0
9
10 while V(l) ≠ ∅ do
11     foreach v ∈ V(l) do
12         if v ∈ S then                                          // c.f. Line 5
13             R_in(v) ← I_i                                          // I_i ∈ 𝕎
14         else
15             P ← {p | (p, v) ∈ E}                                // producers
16             R_in(v) ← Σ_{∀p∈P} R_out(p)                    // Equation 3.16
17
18             b_in(v) ← (Σ_{∀p∈P} R_out(p)) / (Σ_{∀p∈P} R_out(p) / b_out(p))   // Equation 3.21
19
20         C(v) ← Equation 3.3                  // apply cost model from Ch.3
21                                                    // starting with Eq.3.3
22                                  // dependent equations omitted for brevity
23
24         if R_in(v) > C(v) then                // bottleneck detected (Eq.4.3)
25             B ← B ∪ {v}
26
27         R_out(v) ← s · min{R_in(v), C(v)}                       // Equation 4.6
28     l ← l + 1
```

First, the input date rate and input batch size is computed (Lines 12-18). For sources (Line 12), the input data rate is provided by the workload (Line 13). No input batch size is computed because sources do not have an input queue. For all other operators, the input data rate and input batch size is computed as described in Section 3.2.2 (Lines 16-18). (2) In the second step (Line 20), the cost model from Chapter 3 is applied to compute the operator capacity. (3) The actual bottleneck detection step follows in Lines 24-25 based on Definition 43. If an operator is overloaded, it is added to the result set B in Line 25. (4) In the last step, the output data rate of the operator is computed (Line 27). After all operators of one level are processed, the algorithm advances to the next level (Line 28). The main loop (Line 10) checks if the current level contain any operators. This condition is true until all sinks are processed and the level is larger than the level of any operator. For this case, the algorithm terminates.

Lemma 1. *The bottleneck detection algorithm (Algorithm 1) has a runtime complexity of $O(|V| \cdot \delta)$ with δ being the maximum* input degree*, i. e., maximum number of incoming edges, over all operators in the data flow program.*

Proof. Algorithm 1 is initialized with level 0 in Line 8, and the level is incremented exactly once within the main loop (Line 28). Hence, the main loop (Lines 10-28) is executed as many times as levels exists in the data flow program. Furthermore, it is clear from Definition 44 that each operator is contained in exactly one level. Thus, each operator will be processed in exactly one `for`-loop execution of the algorithm. Processing a single operator (Lines 12-27) has runtime complexity $O(\delta)$ because Equation 3.16 (Line 16), Equation 3.21 (Line 18), and Equation 3.27 (Line 20)[2] depend on the number of upstream producers. Hence, the complexity of Algorithm 1 is $O(|V| \cdot \delta)$. □

Corollary 2. *The complexity of Algorithm 1 is $O(|V|^2)$ because in general δ grows with $|V|$. For example, let D be a graph with x sources and x sinks (i. e., $|V| = 2 \cdot x$), and each source is connected to all sinks. For this case, each sink has $|V|/2$ producers, and thus, $\delta = |V|/2$.*

In practice, δ does not grow with $|V|$ but is a constant. Hence, in practice we expect a linear runtime in the number of operators $|V|$ of Algorithm 1.

Algorithm 1 only identifies bottlenecks in the data flow program, but does not categorize them into CPU or network bottlenecks. It is straightforward to extend the algorithm to compute a categorization, and thus, we omit it for brevity. The categorization is useful, because it helps to change a configuration to avoid bottlenecks. For network bottlenecks, it is required to increase the parallelism to increase the operator capacity. For CPU bottlenecks, increasing the output batch size or input batch size[3] of an operator may also increase its capacity. We refer to Section 4.2 for an algorithm that computes a configuration that avoid bottlenecks in a data flow program.

Above, we introduced our bottleneck detection algorithm (Algorithm 1) and defined bottlenecks formally (Definition 43). Next, we introduce an algorithm that predicts the effective throughput of a data flow program for the case that a bottleneck exists.

4.1.2 Throughput Prediction

Detecting bottlenecks as discussed in the previous section is useful to verify if a configuration of a data flow program results in bottlenecks. However, it does not provide any insight into the achieved throughput if a bottleneck exists. Thus, we introduce a second algorithm that computes the achieved throughput per source. In alignment to the data flow capacity (Definition 25) and workload (Definition 42), we define the *data flow throughput* as a vector (c. f. Definition 46 below) that describes the throughput per source operator. Before we formally define the data flow

[2] Equation 3.27 is used within Equation 3.3 to compute the output network capacity C_o.

[3] The input batch size can be increased indirectly, by increasing the output batch sizes of the corresponding producers.

throughput, we introduce *effective* operator input and operator output data rate as follows:

Definition 45 (Effective Operator Input and Output Data Rate)**.** *Given an operator v in a data flow program. The* effective input data rate $\widehat{R}_{\text{in}}$ *of v is the number of input records per second that is processed by v. The* effective output data rate $\widehat{R}_{\text{out}}$ *of v is the number of output records per second that v sends downstream.*

Definition 45 implies that $\widehat{R}_{\text{in}} \leq R_{\text{in}}$ and that $\widehat{R}_{\text{out}} \leq R_{\text{out}}$. We call the effective input data rate of an operator the operator *throughput*. Furthermore, the *data flow throughput* is defined as follows:

Definition 46 (Throughput of a Data Flow Program)**.** *Given a data flow program D with sources $s_1, \ldots, s_n$. The throughput $\mathbb{T}$ of D with regard to a configuration Γ and workload $\mathbb{W}$ is:*

$$\mathbb{T}(D, \Gamma, \mathbb{W}) = \langle \widehat{R}_{\text{in}}(s_1), \ldots, \widehat{R}_{\text{in}}(s_n) \rangle \tag{4.8}$$

Corollary 3 (Throughput and Workload Relationship)**.** *For any given triple D, Γ, and $\mathbb{W}$, Definition 46 implies that:*

$$\mathbb{T} \leq \mathbb{W} \tag{4.9}$$

with

$$\mathbb{T} \leq \mathbb{W} \iff \forall \widehat{R}_{\text{in}}(s_i) \in \mathbb{T} : \widehat{R}_{\text{in}}(s_i) \leq I_i$$

In the following, we discuss how to compute the throughput of a data flow program. Before we introduce an algorithm to compute the data flow throughput, we explain the dependencies of effective input and output data rates (Definition 45) between the operators of a data flow program. If a single operator cannot send its output data in a timely manner, e. g., there is a network bottleneck, the effective output data rate limits the effective input data rate. Furthermore, given a producer p and its consumer c, there is a dependency between $\widehat{R}_{\text{out}}(p)$ and $\widehat{R}_{\text{in}}(c)$, because the producer output data rate may be limited by the effective consumer input data rate. This dependency is called *back pressure* [SGH15, CWI+16, FAG+17, KRK+18]. A slower downstream operator throttles the throughput of upstream operators. In the following, we discuss this dependency and how we compute $\widehat{R}_{\text{in}}$ and $\widehat{R}_{\text{out}}$. We point out that back pressure propagates in the reverse direction to the data flow (i. e., from higher levels to lower levels). Thus, we start our discussion beginning with sinks.

Definition 47 (Effective Sink Input Data Rate)**.** *Let v be a sink in a data flow program. Because sinks do not have any downstream operators, they are not subject to back pressure. Therefore, the effective input data rate of v only depends on its input data rate and capacity and is computed as:*

$$\widehat{R}_{\text{in}} = \min\{R_{\text{in}}, C\} \tag{4.10}$$

If we know the effective input data rate of an operator, we compute the effective output data rates for all its upstream producers as follows:

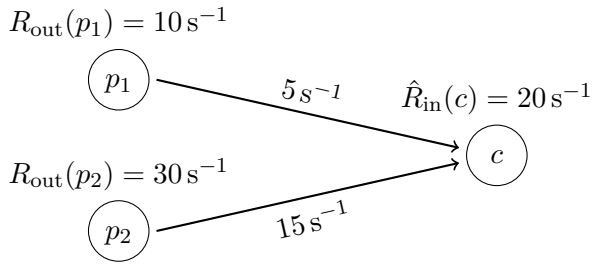

Figure 4.3: Back pressure from consumer c to producers p_1 and p_2.

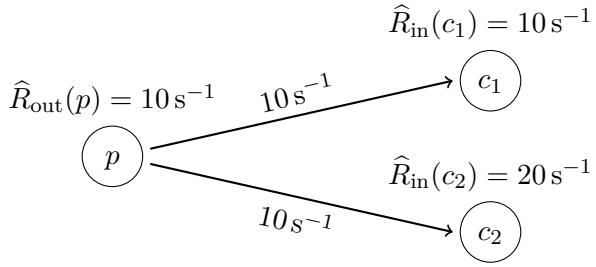

Figure 4.4: Back pressure from consumers c_1 and c_2 to producer p.

Definition 48 (Effective Operator Output Data Rate)**.** *Given a consumer c with producers $p_1, \ldots, p_n$. The effective output data rate for each producer p_i is computed as:*

$$\widehat{R}_{\text{out}}(p_i) = \frac{R_{\text{out}}(p_i)}{R_{\text{in}}(c)} \cdot \widehat{R}_{\text{in}}(c) \tag{4.11}$$

Example 10. *Assume two producers p_1 and p_2 with data rates $R_{\text{out}}(p_1) = 10\,\text{s}^{-1}$ and $R_{\text{out}}(p_2) = 30\,\text{s}^{-1}$ and a consumer c with an effective input data rate $\widehat{R}_{\text{in}}(c) = 20\,\text{s}^{-1}$ (Figure 4.3). The data rate of $20\,\text{s}^{-1}$ that c can process, is distributed between p_1 and p_2. Because p_1 sends $25\% = \frac{10\,\text{s}^{-1}}{40\,\text{s}^{-1}} = \frac{R_{\text{out}}(p_1)}{R_{\text{in}}(c)}$ of all input data (i. e., $R_{\text{in}}(c) = 10\,\text{s}^{-1} + 30\,\text{s}^{-1} = 40\,\text{s}^{-1}$), it can effectively send $\widehat{R}_{\text{out}}(p_1) = 25\% \cdot \widehat{R}_{\text{in}}(c) = 25\% \cdot 20\,\text{s}^{-1} = 5\,\text{s}^{-1}$. Similarly, p_2 contributes $75\% = \frac{30\,\text{s}^{-1}}{40\,\text{s}^{-1}}$ of $R_{\text{in}}(c)$, and thus, can effectively send $\widehat{R}_{\text{out}}(p_2) = 75\% \cdot 20\,\text{s}^{-1} = 15\,\text{s}^{-1}$.*

Example 10 illustrates how back pressure propagates from downstream to upstream operators. If downstream back pressure occurs, it limits the throughput, i. e., effective input data rate (Definition 45), of an operator. For this case, we compute the effective input data rate as follows:

Definition 49 (Effective Operator Input Data Rate)**.** *Given a producer p with selectivity s (c. f. Section 3.2) and consumers $c_1, \ldots, c_n$. Each consumer c_i might limit the producer output data rate, and thus, the lowest $\widehat{R}_{\text{in}}(c_i)$ determines $\widehat{R}_{\text{out}}(p)$. Hence, we compute $\widehat{R}_{\text{in}}$ of p as:*

$$\widehat{R}_{\text{in}}(p) = \frac{\widehat{R}_{\text{out}}(p)}{s} \tag{4.12}$$

with

$$\widehat{R}_{\text{out}}(p) = \min\{\widehat{R}_{\text{in}}(c_i) | 1 \leq i \leq n\} \tag{4.13}$$

Algorithm 2: Throughput Prediction

```
1  Input: data flow D = (V, E); configuration Γ(D); workload 𝕎(D)
2  Output: throughput 𝕋(D, Γ, 𝕎)
3
4  // compute capacity, input, and output data rates
5  bottleneckDetection(D,Γ,𝕎)                          // use Algorithm 1
6
7  S̄ ← {v ∈ V | ∄v' : (v, v') ∈ E}                                  // sinks
8  l ← l − 1                    // initialized in Line 5 via Algorithm 1
9
10 // compute effective input and output data rates
11 while l ≥ 0 do
12     foreach v ∈ V(l) do
13         if v ∈ S̄ then
14             R̂_in(v) ← min{R_in(v), C(v)}                  // Equation 4.10
15         else
16             R̂_out(v) ← min{ (R_out(v)/R_in(c)) · R̂_in(c) | (v, c) ∈ E }   // Equation 4.11
17
18             R̂_in(v) ← min{R̂_in(c) | (v,c) ∈ E} / s                 // Equation 4.12
19     l ← l − 1
```

Example 11. *Assume a producer p with two consumers c_1 and c_2 with maximum input data rates $\widehat{R}_{\text{in}}(c_1) = 10\,\text{s}^{-1}$ and $\widehat{R}_{\text{in}}(c_2) = 20\,\text{s}^{-1}$ (Figure 4.4). The maximum output data rate of p is limited by the smallest maximum input data rate of its consumers, and thus, $\widehat{R}_{\text{out}}(p) = 10\,\text{s}^{-1}$.*

Given Equation 4.10, Equation 4.11, and Equation 4.12 we can compute the throughput of a data flow program as follows: first, we identify the bottleneck operators. We emphasize that there might be multiple bottlenecks on a path from a source to a sink, and thus, the bottlenecks in higher levels of the data flow program determine the data flow throughput. After those bottlenecks are detected, we backtrack to the sources using the equations from above to compute the effective input and output data rate for each operator.

Algorithm 2 computes the throughput of a data flow program. In the first phase it uses Algorithm 1 to compute the operator capacity as well as the operator input and output data rate for all operators in the data flow program (Line 5). In the second phase (Lines 11-19), the data flow graph is traversed in the reverse order (Line 19), starting at the largest level (Line 8) and terminating after level 0 was processed (Line 11). In this phase, the effective input and output data rates for each operator are computed. Finally, the algorithm returns the data flow throughput that comprises the effective source input data rates.

Corollary 4. *The runtime complexity of Algorithm 2 is the same as the runtime complexity of Algorithm 1. Equation 4.10 and Equation 4.11 have constant runtime and Equation 4.12 has complexity $O(\delta)$. Furthermore, each operator is processed exaclty once in the second phase of the algorithm, similarly to the first phase.*

In this section, we introduced algorithms that take a data flow program and its configuration as input and compare the computed data flow capacity to a given workload. These algorithms are useful to reason about a given configuration. However, it implies that a user needs to specify the configuration manually. In the next section, we introduce an algorithm that computes a configuration, lifting the requirement for the user to specify a configuration.

4.2 Minimizing Resource Consumption

A common goal in data flow provisioning is to minimize the used compute resources while avoiding bottlenecks at the same time. To this end, we employ the following resource model. We assume a homogeneous cluster of servers onto which the stream processing system is deployed. Given a server with n CPU cores and a server network bandwidth of $\hat{\mathcal{N}}$ (Section 3.3), we assign n tasks slots per server. Each task slot may executed a single task using a single thread for execution. Hence, a task slot may utilize a single CPU core. The available network bandwidth is shared over all n tasks slots, i. e., each task slot has a network bandwidth of $\mathcal{N} = \hat{\mathcal{N}}/n$. We assume that a task will fully utilizes either its CPU or its network resource. Hence, our resource model ensures that a single server is never overloaded because we allow for n task slots per server. Because a task may not fully utilize both its CPU and network capacity, using n tasks slots may result in some under utilization of a server. We consider this an orthogonal scheduling problem that is beyond the scope of this thesis, and refer to related work (Section 4.4).

In Section 3.1, we discussed that batching can increase the capacity of an operator. Furthermore, for a fixed input data rate (i. e., given workload), an increased capacity allows to run fewer tasks. The required resources to execute a data flow program depend on the number of deployed tasks. Hence, to minimize the required resources, we want to find a configuration that uses batch sizes that allows to deploy a minimum number of tasks.

Before we introduce our optimization goal formally, we define the *resource consumption* of a data flow program with respect to its configuration as follows:

Definition 50 (Resource Consumption)**.** *Let $D = (V, E)$ be a data flow program with configuration Γ resulting in execution graph $EG = (T, F)$ (Definition 4). The* resource consumption *RC of EG is:*

$$RC(EG) = |T| \tag{4.14}$$

The resource consumption can also be expressed as a function of D and Γ:

$$RC(D, \Gamma) = \sum_{v \in V} dop(v) \tag{4.15}$$

The resource consumption is the number of all tasks of an execution graph, which equals to the sum of all *dop* values over all operators in D with respect to Γ. Because we assume that each task is deployed into one task slot as discussed above, RC implicitly describes the required number of CPU cores and network consumption that are required to execute EG.

To compute a configuration that minimizes the used resources, we need to find a configuration that minimizes the overall parallelism. At the same time, this configuration must result in a data flow capacity (Definition 25) that avoids bottlenecks, as defined below:

Definition 51 (Workload-Capacity-Relationship)**.** *Given a workload* $\mathbb{W} = \langle I_1, \ldots, I_n \rangle$ *(Definition 42), the set of all capacities* $\mathbb{C}^\dagger$*, and a capacity* $\mathbb{C} \in \mathbb{C}^\dagger$ *for a data flow program* D *with configuration* Γ*. We say* $\mathbb{C} = \langle c_1, \ldots, c_n \rangle$ *is equal or greater than* $\mathbb{W}$*, denoted* $\mathbb{C} \geq \mathbb{W}$*, iff:*

$$\mathbb{C} \geq \mathbb{W} \iff \forall c_i \in \mathbb{C} : c_i \geq I_i \tag{4.16}$$

Definition 51 states that a capacity is larger than a workload if all capacity elements are larger than their corresponding input data rates of the workload. The definition of data flow capacity (Definition 25) implies that no operator in D is overloaded for this case—otherwise $\mathbb{C}$ would violate its definition.

Using Definition 50 and Definition 51 we describe the problem to minimize the resource consumption of an execution graph (given a data flow program and workload), such that there is no bottleneck, as the following minimization problem:

Definition 52 (Minimizing Resource Consumption)**.** *Given a data flow program* $D = (V, E)$ *and a workload* $\mathbb{W}$*. Find a configuration* Γ *with:*

$$\begin{aligned} &\min_{\forall \Gamma(V)} \{RC(D, \Gamma)\} \\ &\textit{with } \exists \mathbb{C} \in \mathbb{C}^\dagger : \mathbb{C} \geq \mathbb{W} \end{aligned} \tag{4.17}$$

The data flow capacity $\mathbb{C}$ depends on the configuration Γ of D. To minimize the resource consumption of D, we determine the *optimal configuration* Γ^* such that RC is minimized by Γ^* and there exists a capacity $\mathbb{C}$ that is larger than the given workload. We emphasize that there is a set of solutions for Γ^*, because our objective function is only required to minimize resource consumption. The batch size for each operator of D might have different values for different configurations in the solution space. Because batch sizes influence processing latency, our algorithm preferably chooses small batch sizes to achieve a low latency following a best effort approach. Using small batch sizes to keep latency small is not part of the optimization problem. Minimizing the latency would require to model the expected latency based on batch sizes. Such a latency model is beyond the scope of this thesis and is interesting future work.

4.2.1 Minimizing Parallelism

The goal of our optimization is to avoid bottlenecks as defined in Definition 43. Thus, we compute a *dop* for each operator such that the operator capacity is larger than the operator input data rate. The capacity of an operator depends on the capacity of the corresponding tasks as well as the number of tasks for this operator. Thus, given the input data rate R_{in} of an operator v we derive that there is no bottleneck for a large enough operator *dop*. Using Equation 4.3 and Equation 3.3

we express this relationship formally as follow:

$$\begin{aligned} & R_{\text{in}}(v) \leq C(v) \\ \iff & R_{\text{in}}(v) \leq \mathit{dop} \cdot C(t^v) \\ \iff & \mathit{dop} \geq \tfrac{R_{\text{in}}(v)}{C(t^v)} \end{aligned} \tag{4.18}$$

Our optimization goal is to minimize RC, which implies to minimize the dop of each operator. Hence, the minimum dop^* of an operator that avoids a bottleneck is:

$$\mathit{dop}^*(v) = \left\lceil \frac{R_{\text{in}}(v)}{C(t^v)} \right\rceil \tag{4.19}$$

Equation 4.19 is based on a given input data rate R_{in} and the task capacity C. If we maximize the task capacity by computing corresponding batch sizes (Section 4.2.2 below), we minimize the dop of an operator. The task capacity is based on the processing capacity, the input network capacity, and the output network capacity (Definition 28):

$$C = \min\{C_{\text{i}}, C_{\text{p}}, C_{\text{o}}\}$$

All parameters to compute the network input capacity and the network output capacity of a task are known. We emphasize that the network capacities are independent of the data flow configuration. However, the input batch size and the output batch size that influence the processing capacity are not given and must be computed by our algorithm. Hence, the processing capacity C_{p} is unknown (Equation 3.5 and Equation 3.8):

$$C_{\text{p}} = \frac{1}{\frac{c_{\text{fetch}}}{b_{\text{in}}} + c_{\text{cpu}} + \frac{s \cdot c_{\text{emit}}}{b_{\text{out}}}}$$

However, to minimize the dop, it is not required to know the processing capacity. Instead, we use the *maximum* processing capacity $\widehat{C}_{\text{p}}$ (Equation 3.11), which is independent of input and output batch size:

$$\widehat{C}_{\text{p}} = \frac{1}{c_{\text{cpu}}}$$

To compute the minimum dop^* of an operator, we take into account that $\widehat{C}_{\text{p}}$ is a strict (i. e., exclusive) bound. $\widehat{C}_{\text{p}}$ can never be achieved because using an infinite batch size is not possible.

Example 12. *Assume an input data rate* $R_{\text{in}} = 10\,\text{s}^{-1}$ *and CPU cost* $c_{\text{cpu}} = 1\,\text{s}$. *If* $\widehat{C}_{\text{p}}$ *could be achieved, it would be sufficient to execute* 10 *tasks assuming enough network capacity. However, the processing cost (Equation 3.8) includes some non-zero input fetch and output emit cost. Hence, the maximum processing capacity* $\widehat{C}_{\text{p}}$ *of* $1\,\text{s}^{-1}$ *is larger than the actual processing capacity, for all input/output batch sizes. Using a degree of parallelism (c. f. Equation 4.19)*

$$\mathit{dop} = \left\lceil \frac{R_{\text{in}}}{\widehat{C}_{\text{p}}} \right\rceil = 10$$

would not be sufficient to avoid a CPU bottleneck.

Using the maximum processing capacity of a task, the minimum dop^* for an operator that avoids a CPU bottleneck is computed as follows:

$$dop^*_{\text{cpu}} = \left\lfloor \frac{R_{\text{in}}}{\widehat{C}_{\text{p}}} \right\rfloor + 1 \tag{4.20}$$

Equation 4.20 incorporates that the maximum processing capacity is an exclusive bound. Applying it to Example 12 results in 11 tasks instead of 10 (as shown originally). We generalize the relationship of Equation 4.19 and Equation 4.20 in the following corollary:

Corollary 5. *For any $\epsilon > 0$, $\epsilon \to 0$, and $C_{\text{p}} = \widehat{C}_{\text{p}} - \epsilon$ it holds that:*

$$\left\lceil \frac{R_{\text{in}}}{C_{\text{p}}} \right\rceil = \left\lfloor \frac{R_{\text{in}}}{\widehat{C}_{\text{p}}} \right\rfloor + 1 \tag{4.21}$$

Equation 4.20 ensures that finite input and output batch sizes exist such that the processing capacity is larger than the input data rate. We discuss how input and output batch size are computed in Section 4.2.2. Using the discussed equations from above, we compute the minimum degree of parallelism dop^* of an operator considering each capacity individually. Since all three capacities are independent from each other, each determines an independent lower bound for the required dop. The minimum dop^* of an operator is the maximum over all three lower bounds. Hence, we split Equation 4.19 in three parts and take the maximum over all three:

$$\begin{aligned} dop^*(v) &= \left\lceil \frac{R_{\text{in}}(v)}{C} \right\rceil \\ &= \left\lceil \frac{R_{\text{in}}(v)}{\min\{C_{\text{i}}, C_{\text{p}}, C_{\text{o}}\}} \right\rceil \\ &= \left\lceil \max\left\{ \frac{R_{\text{in}}(v)}{C_{\text{p}}}, \frac{R_{\text{in}}(v)}{C_{\text{i}}}, \frac{R_{\text{in}}(v)}{C_{\text{o}}} \right\} \right\rceil \\ &= \max\left\{ \left\lceil \frac{R_{\text{in}}(v)}{C_{\text{p}}} \right\rceil, \left\lceil \frac{R_{\text{in}}(v)}{C_{\text{i}}} \right\rceil, \left\lceil \frac{R_{\text{in}}(v)}{C_{\text{o}}} \right\rceil \right\} \end{aligned} \tag{4.22}$$

As a last step, we use Equation 4.21 to substitute the unknown processing capacity by the maximum processing capacity:

$$dop^*(v) = \max\left\{ \left\lfloor \frac{R_{\text{in}}(v)}{\widehat{C}_{\text{p}}} \right\rfloor + 1, \left\lceil \frac{R_{\text{in}}(v)}{C_{\text{i}}} \right\rceil, \left\lceil \frac{R_{\text{in}}(v)}{C_{\text{o}}} \right\rceil \right\} \tag{4.23}$$

We use Equation 4.23 in our optimization algorithm, to minimize the parallelism for each operator. To ensure that there is no bottleneck for the computed dop^*, we compute corresponding batch sizes as discussed in the next section.

4.2.2 Batch Size Computation

In the previous section, we discussed how we compute the minimum dop^* of each operator for a given data flow program and workload. To find a solution for our optimization problem (Equation 4.17), we must compute an output batch size for each

operator such that there is no bottleneck in the execution graph of D. Since batching increases processing latency it is desired to use small batch sizes to reduce the processing latency. Our algorithm does not guarantee a minimum processing latency but uses a best effort approach to compute small batch sizes for each operator.

Considering our discussion from Section 4.2.1 we know that dop^* is large enough to avoid network bottlenecks. To ensure that dop^* also avoids CPU bottlenecks, the batch sizes must be large enough such that the processing capacity is larger than the task input data rate (Equation 3.5 and Equation 3.8):

$$\begin{aligned} r_{\text{in}} &\leq C_{\text{p}} \\ &\leq \frac{1}{c_{\text{p}}} \\ &\leq \frac{1}{\frac{c_{\text{fetch}}}{b_{\text{in}}} + c_{\text{cpu}} + \frac{s \cdot c_{\text{emit}}}{b_{\text{out}}}} \end{aligned} \tag{4.24}$$

From Equation 4.24 we derive that there are lower bounds for b_{in} and b_{out} in the solution space. Increasing only the input batch size or only the output batch size might not be sufficient to achieve the required processing capacity.

Example 13. *Let v be an operator with costs $c_{\text{fetch}} = c_{\text{emit}} = c_{\text{cpu}} = 900\,\text{ms}$ and a selectivity $s = 1$. Assuming an input data rate of $R_{\text{in}} = 1\,\text{s}^{-1}$, we derive based on Equation 4.20 and Equation 3.11 that a single tasks is sufficient to avoid a CPU bottleneck:*

$$dop^* = \left\lfloor \frac{R_{\text{in}}}{\frac{1}{c_{\text{cpu}}}} \right\rfloor + 1 = \left\lfloor \frac{1\,\text{s}^{-1}}{\frac{1}{900\,\text{ms}}} \right\rfloor + 1 = \left\lfloor \frac{900\,\text{ms}}{1000\,\text{ms}} \right\rfloor + 1 = 1$$

Let $b_{\text{in}} = 5$: For this case, if $b_{\text{out}} \to \infty$, the minimum processing cost is $1080\,\text{ms}$:

$$\begin{aligned} c_{\text{p}} &= \lim_{b_{\text{out}} \to \infty} \frac{c_{\text{fetch}}}{b_{\text{in}}} + c_{\text{cpu}} + \frac{s \cdot c_{\text{emit}}}{b_{\text{out}}} \\ &= \lim_{b_{\text{out}} \to \infty} \frac{900\,\text{ms}}{5} + 900\,\text{ms} + \frac{1 \cdot 900\,\text{ms}}{b_{\text{out}}} \\ &= 180\,\text{ms} + 900\,\text{ms} + 0\,\text{ms} = 1080\,\text{ms} \end{aligned}$$

Thus, a CPU bottleneck exists because the processing capacity is $C_{\text{p}} = {}^1/_{1080\,\text{ms}} < R_{\text{in}}$. Only if the input batch size is at least 10, an output batch size of 90 or higher is sufficient to avoid a CPU bottleneck:

$$c_{\text{p}} = \frac{900\,\text{ms}}{10} + 900\,\text{ms} + \frac{900\,\text{ms}}{90} = 1000\,\text{ms}$$

Some other solutions (i. e., batch sizes that avoid a CPU bottleneck) for this example are $b_{\text{in}} = 15$ and $b_{\text{out}} = 25$, or $b_{\text{in}} = 20$ and $b_{\text{out}} = 20$.

Example 13 illustrates that there are lower bounds for input and output batch size and that there are multiple solutions for input and output batch sizel to avoid

a CPU bottleneck. We compute the lower bounds b_{in}^* and b_{out}^* of b_{in} and b_{out} based on Equation 4.24 as follows:

$$\begin{aligned} b_{\text{in}} &\geq \frac{c_{\text{fetch}}}{\frac{1}{r_{\text{in}}} - c_{\text{cpu}} - \frac{s \cdot c_{\text{emit}}}{b_{\text{out}}}} \\ \implies b_{\text{in}}^* &= \left\lfloor \lim_{b_{\text{out}} \to \infty} \frac{c_{\text{fetch}}}{\frac{1}{r_{\text{in}}} - c_{\text{cpu}} - \frac{s \cdot c_{\text{emit}}}{b_{\text{out}}}} \right\rfloor + 1 \\ &= \left\lfloor \frac{c_{\text{fetch}}}{\frac{1}{r_{\text{in}}} - c_{\text{cpu}}} \right\rfloor + 1 \end{aligned} \tag{4.25}$$

and

$$\begin{aligned} b_{\text{out}} &\geq \frac{s \cdot c_{\text{emit}}}{\frac{1}{r_{\text{in}}} - c_{\text{cpu}} - \frac{c_{\text{fetch}}}{b_{\text{in}}}} \\ \implies b_{\text{out}}^* &= \left\lfloor \lim_{b_{\text{in}} \to \infty} \frac{s \cdot c_{\text{emit}}}{\frac{1}{r_{\text{in}}} - c_{\text{cpu}} - \frac{c_{\text{fetch}}}{b_{\text{in}}}} \right\rfloor + 1 \\ &= \left\lfloor \frac{s \cdot c_{\text{emit}}}{\frac{1}{r_{\text{in}}} - c_{\text{cpu}}} \right\rfloor + 1 \end{aligned} \tag{4.26}$$

A solution for our optimization problem (Equation 4.17) is a configuration that contains output batch sizes, but no input batch sizes. To use Equation 4.24 to compute output batch sizes, we first need to compute the corresponding input batch size b_{in}. Input batch sizes depend on the upstream producer output batch sizes (c. f. Corollary 1). To resolve mutual dependencies, we exploit the structure of a data flow program: (1) sources do not have input batch sizes, and thus, we can compute their output batch sizes; (2) the operator levels (Definition 44) allow us to compute b_{in} before b_{out} for all other nodes. Knowing b_{in}, we compute b_{out} based on Equation 4.24 as follows:

$$b_{\text{out}} = \left\lceil \frac{s \cdot c_{\text{emit}}}{\left(\frac{1}{r_{\text{in}}} - c_{\text{cpu}} - \frac{c_{\text{fetch}}}{b_{\text{in}}}\right)} \right\rceil \tag{4.27}$$

To use Equation 4.27 we consider the following: for a source node there is no input queue, and thus, we set $c_{\text{fetch}} = 0$ that allows us to compute a minimum output batch size without b_{in} as a parameter. If we know the output batch sizes of a producer (e. g., a source), we can compute corresponding input batch sizes for their consumers from the next level. For this step, the effective input batch size must be larger then the minimum input batch size. If this condition is violated, we must increase the producer output batch sizes accordingly. After the effective input batch sizes are known, we compute the output batch size of downstream nodes. Following this pattern, it is possible to compute an output batch size for all operators in a data flow program level-by-level. We describe an algorithm that uses this approach in more detail in the next section.

4.2.3 Algorithm Resource Optimizer

In this section, we introduce a resource minimization algorithm (Algorithm 3) that computes a solution for the optimization problem as specified in Definition 52. Algorithm 3 takes a data flow program D and a workload $\mathbb{W}(D)$ as input and computes

Algorithm 3: Resource Minimizer

1 **Input:** data flow: $D = (V, E)$; workload: $\mathbb{W}(D)$
2 **Output:** configuration Γ that is a solution to Equation 4.17
3
4 Def: $V(l) \leftarrow \{v \in V | l(v) = l\}$ `// all operators of level l`
5 $S \leftarrow V(0)$ `// get sources`
6
7 $l \leftarrow 0$ `// start at level 0`
8 **while** $V(l) \neq \emptyset$ **do**
9 **foreach** $v \in V(l)$ **do**
10 **if** $v \in S$ **then** `// c.f. Line 5`
11 $R_{\text{in}}(v) \leftarrow I_i$ `//` $I_i \in \mathbb{W}$
12 **else**
13 $R_{\text{in}}(v) \leftarrow \sum_{\forall p \in V : \exists (p,c) \in E} R_{\text{out}}(p)$ `// Equation 3.16`
14
15 $dop(v) \leftarrow \max \left\{ \left\lfloor \frac{R_{\text{in}}(v)}{\widehat{C}_{\text{p}}} \right\rfloor + 1, \left\lceil \frac{R_{\text{in}}(v)}{C_{\text{i}}} \right\rceil, \left\lceil \frac{R_{\text{in}}(v)}{C_{\text{o}}} \right\rceil \right\}$ `// Equation 4.23`
16
17 **if** $v \notin S$ **then** `// c.f. Line 5`
18 $b_{\text{in}}(v) \leftarrow \frac{\sum_{i=1}^{n} R_{\text{out}}(p_i)}{\sum_{i=1}^{n} \frac{R_{\text{out}}(p_i)}{b_{\text{out}}(p_i)}}$ `// Equation 3.21`
19 $b^*_{\text{in}}(v) \leftarrow \left\lfloor \frac{c_{\text{fetch}}}{\frac{1}{r_{\text{in}}} - c_{\text{cpu}}} \right\rfloor + 1$ `// Equation 4.25`
20
21 **while** $b_{\text{in}}(v) < b^*_{\text{in}}(v)$ **do**
22 $\bar{p} \leftarrow p \in V : (p, v) \in E \wedge$
23 $b_{\text{out}}(p) = \min\{b_{\text{out}}(p') | \forall p' \in V : \exists (p', v) \in E\}$
24 $b_{\text{out}}(\bar{p}) \leftarrow b^*_{\text{in}}(v)$
25 $b_{\text{in}}(v) \leftarrow \frac{\sum_{i=1}^{n} R_{\text{out}}(p_i)}{\sum_{i=1}^{n} \frac{R_{\text{out}}(p_i)}{b_{\text{out}}(p_i)}}$ `// Equation 3.21`
26
27 **if** $\exists v' \in V : (v, v') \in E$ **then** `// non-sinks`
28 $b_{\text{out}}(v) \leftarrow \left\lceil \frac{s \cdot c_{\text{emit}}}{\left(\frac{1}{r_{\text{in}}} - c_{\text{cpu}} - \frac{c_{\text{fetch}}}{b_{\text{in}}(v)} \right)} \right\rceil$ `// Equation 4.27`
29 $R_{\text{out}}(v) \leftarrow s \cdot R_{\text{in}}(v)$ `// Equation 3.15`
30 $l \leftarrow l + 1$

a configuration Γ for D that is a solution to Equation 4.17. Algorithm 3 is based on the equations from the previous sections.

Algorithm 3 traverses the data flow program based on levels (similarly to Algorithm 1 and Algorithm 2). For each operator of a level (Line 9), the algorithm computes *dop* and b_{out} as follows: first it computes the operator input data rate. For sources, the input data rate is the same as provided by the workload that is an input to the algorithm (Line 11). For all other operators, the input data rate depends on the upstream operator output data rates (Line 13). In Line 15, the algorithm

computes the minimum *dop** as described in Section 4.2.1. If an operator is not a source, Algorithm 3 computes the operator input batch size following Section 4.2.2 (Lines 17-25). First, it computes the operator input batch size based on the upstream output batch sizes (Line 18). The input batch size must not be smaller than the minimum input batch size b_{in}^* that is computed in Line 19. As long as $b_{\text{in}} < b_{\text{in}}^*$ (Lines 21-25), the algorithm increases the output batch of the producer with the smallest output batch size (Line 22) to b_{in}^* (Line 24). In Line 25, the input batch size is recomputed using the updated upstream output batch size. The last phase of the algorithm does not apply to sink nodes (Lines 27-29) because those do not have output queues or downstream consumers. In this phase, the algorithm computes output batch sizes (Line 28) and output data rates (Line 29). We use Equation 3.15 instead of Equation 4.6 because we know that no bottleneck exists. After all operators of a level are processed, Algorithm 3 advances to the next level (Line 30) until all operators are processed (i. e., $V(l) = \emptyset$) and the algorithm terminates (Line 8).

Lemma 2. *Algorithm 3 computes a configuration that is a solution to the "Minimizing Resource Consumption" optimization problem (Definition 52).*

Proof. Algorithm 3 computes a configuration that minimizes *RC* because it computes a minimal *dop* (Line 15) for each operator of the input data flow program. Since the *dop* of each operator is minimized, it follows that *RC* is minimized. If only $RC - 1$ tasks are deployed, one operator will not have enough capacity to process its input data rate and would be a bottleneck. Furthermore, Algorithm 3 computes output batch sizes such that no operator is a CPU bottleneck. For sources, this follows from our cost model (Line 28). For all other operators Lines 21-25 ensure that $b_{\text{in}} \geq b_{\text{in}}^*$. If b_{in} is smaller that b_{in}^* at least one upstream b_{out} must be smaller than b_{in}^*, because b_{in} is computed as a weighted average over all upstream b_{out} values (Equation 3.21). Therefore, setting $b_{\text{out}}(p)$ to b_{in}^* (Line 24) increases $b_{\text{out}}(p)$ and it follows that b_{in} is increased (Line 25). The `while`-loop is guaranteed to terminate, because for all b_{out} it holds that it is either larger than b_{in}^* initially, or it may be increased to b_{in}^* ensuring that the weighted average over all b_{out} will eventually be at least b_{in}^*. Since b_{in} will be at least b_{in}^* after the `while`-loop it follows that Line 28 computes an output batch size that avoids a CPU bottleneck. Last, increasing the output batch size of upstream producers (Line 24) can only increase the producer task capacity, and thus, cannot introduce a bottleneck. □

Lemma 3. *The complexity of Algorithm 3 is $O(|V| \cdot \delta^2)$, with δ being the maximum input degree over all operators in the data flow program.*

Proof. Algorithm 3 executes the outer `while`-loop (Lines 8-30) once for each level (Line 30). The `for`-loop (Lines 9-29) is executed once per operator per level. Hence, overall it is executed exactly once per operator, because each operator is assigned to exactly one level. The inner `while`-loop (Lines 21-25) is executed at most once per incoming edge of v and has linear runtime in δ because the input batch size is recomputed in Line 25. Hence, the inner `while`-loop has a runtime complexity of $O(\delta^2)$. All other lines have constant or linear runtime in δ. Hence, the overall runtime complexity of Algorithm 3 is $O(|V| \cdot \delta^2)$. □

Even if the complexy of Algorithm 3 is $O(|V| \cdot \delta^2) = O(|V|^3)$, in practice, δ does not grow with $|V|$ but is a constant (c. f. Corollary 2). Hence, in practice we expect a linear runtime in the number of operators $|V|$ of Algorithm 3.

4.3 Evaluation

In the previous chapter (Chapter 3), we introduced our cost model and discussed different batching techniques (Section 3.4). Using our cost model, we introduced several algorithms to detect bottlenecks (Section 4.1.1), to predict the data flow throughput (Section 4.1.2), and to minimize resource consumption (Section 4.2). In this section, we present results derived from our cost model and algorithms.

We first present some micro benchmark results that show the impact of batching on the capacity/throughput of tasks. Later, we evaluate the performance of data flow programs for different configurations. For this evaluation, we are using Apache Storm [ASFf, TTS+14] an open-source, distributed, and data-parallel stream processing system. Apache Storm does not support batching natively; thus, we implemented a custom batching layer using the shared batching scheme as described in Section 3.4.

We ran our experiments using Apache Storm version 0.9.3 and Open-JDK 8 (64-bit) with Ubuntu Server 16.04 LTS (64-bit). Our processing cluster consists of 28 machines each equipped with two Xeon E5-2620 2 GHz CPUs (6 cores) and 24 GB of main memory. The machines are connected via 10 Gbit Ethernet. The clocks of all machines are synchronized via NTP.

Before we describe our experiments, we discuss some details about our batching layer implementation in Apache Storm:

Transparent Batching Implementation Our approach in implementing batching is non-intrusive, i. e., we do not alter the streaming system.[4] Non-intrusive batching has the advantage that a user does not need to install a modified version of Apache Storm. We built our library in such a way that it can be used without even altering the existing user code, in particular user-defined functions. Hence, our batching layer is transparent both to the system and to the user. We implemented our batching layer as user-defined second-order functions that take the original user functions as input. Our second-order functions are a wrapper to batch result records, and a wrapper to de-batch incoming records. A batch is basically a fat record—we call it *batch record*. To Storm, a batch record looks like a regular (but very big) record. Because Storm performs some sanity-checks on emitted records, we used a *vertical layout* for our batch records. A straightforward batch layout that concatenates multiple records into a single record would not work with Storm, because Storm expects the number of attributes of output records to match a user-defined schema. For example, using a batch layout with a single attribute (that is a concatenation/list of records) or a "wide" schema (a concatenation of all attributes of all records within a batch) would be rejected by Storm, because the actual and expected number of attributes would not match. Hence, we implemented a vertical batch layout similar to the column-based table layout as used in some relational database

[4] The code is available on GitHub: `https://github.com/mjsax/aeolus`.

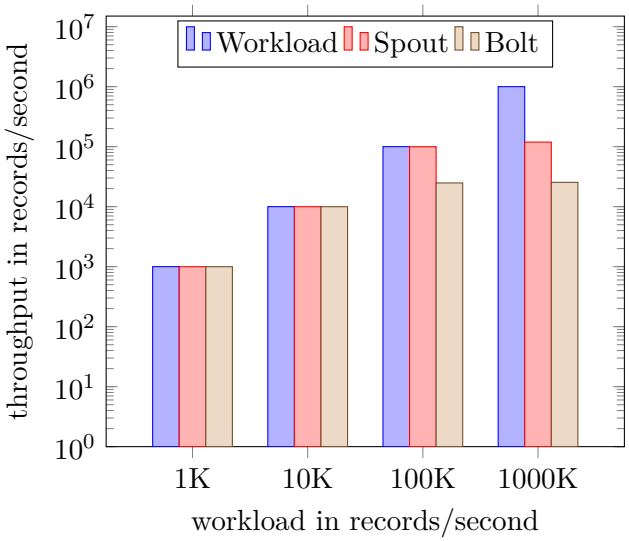

Figure 4.5: Spout/bolt throughput for $b_{\text{out}} = 1$ and different workloads.

systems. The schema of a batch record comprises list-attributes. Each user record is split into its attributes and the attributes are appended to the corresponding list-attribute of the batch record. For example, three records $r_1 = \langle a_1, b_1 \rangle$, $r_2 = \langle a_2, b_2 \rangle$, and $r_3 = \langle a_3, b_3 \rangle$ are inserted into a batch record b as $\langle [a_1, a_2, a_3], [b_1, b_2, b_3] \rangle$. Therefore, the batch record has the same number of attributes as the original records and Storm accepts the batch record (Storm does not check the attributes types).

Additionally, it is necessary to change the hash function used by Storm, i. e., the default `.hashCode()` method that all Java classes have. For each record, Storm calls `.hashCode()` on the key attributes to determine the consumer task per node. Therefore, we cannot use Java's standard `List` implementations to represent a column of the batch record because it would change the hash value inconsistently. The `List.hashCode()` implementation considers all elements in the list and computes a different hash value compared to the hash value of any single value in the list. Hence, we use a custom `List` implementation (`AttributeList` class) that returns the hash value of the first element in the list. We could use any element in the list because we know that all elements have the same hash value modulo number of consumer tasks—otherwise a record would have been inserted into a different buffer (c. f. Section 3.4). Therefore, `AttributeList.hashValue()` ensures that the hash value modulo number of consumer tasks of a record batch is the same as for each record in the record batch. Using `AttributeList` ensures that the partitioning stays the same when Storm computes the target task for a single record or batch.

4.3.1 Throughput

In this section, we evaluate the impact of batching on task capacity, and hence, task throughput. We first consider a single consumer-producer pair for a base-line evaluation. Later, we consider more complex data flow programs with multiple different batch sizes.

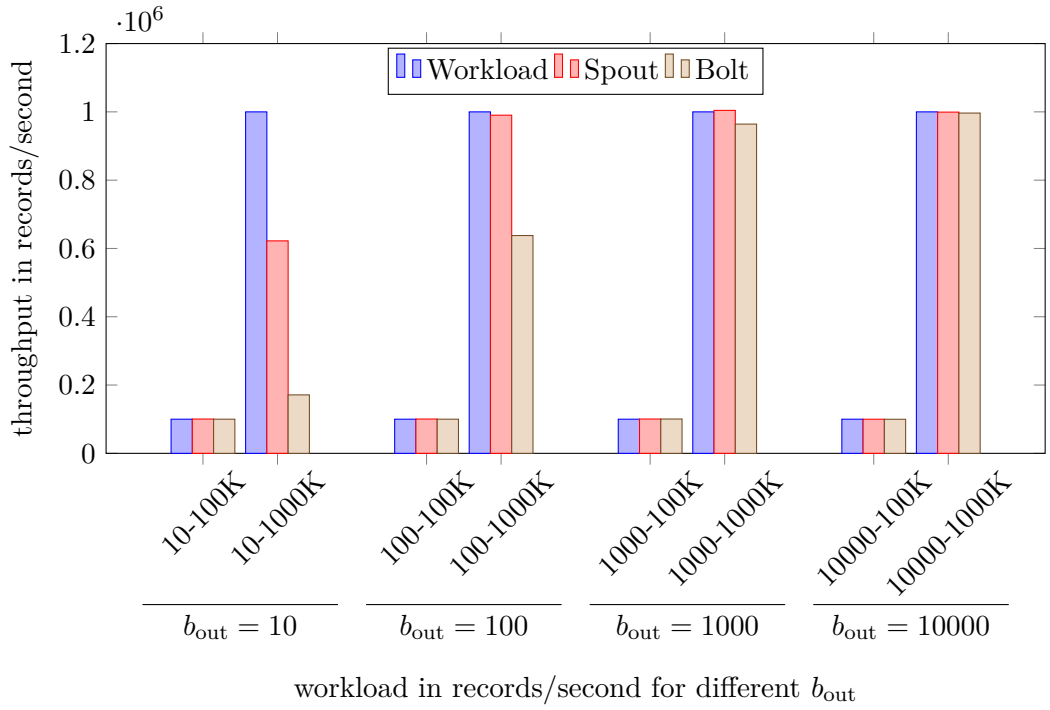

Figure 4.6: Spout/bolt throughput for different batch sizes and workloads.

Batch Size and Throughput

Our first experiment is a micro-benchmark that measures the throughput of a source and its consumer. Sources are called *spouts*, and all other nodes are called *bolts* in Storm. A data flow program, comprising spouts and bolts, is called a *topology*. Hence, our first topology consists of a single spout and a single bolt that are connected to each other. We do not consider parallelism in this experiment, and execute both operators with $dop = 1$. The spout generates random data in-memory (with a record size of 100 B), and we vary the data rate (i. e., workload) as well as the spout output batch size.

Figure 4.5 depicts the observed throughput numbers for workloads of $1000\,\mathrm{s}^{-1}$ to $1\,000\,000\,\mathrm{s}^{-1}$, and a spout output batch size of $b_{\text{out}} = 1$ (i. e., no batching). Hence, Figure 4.5 presents our base line result, and we discuss the performance impact of batching in the next paragraph. Because there is a single spout and a single bolt, the bolt input batch size is $b_{\text{in}} = 1$. For small data rates of $1000\,\mathrm{s}^{-1}$ and $10\,000\,\mathrm{s}^{-1}$ the spout throughput and the bolt throughput are equal to the workload indicating that there is no bottleneck in the topology. However, for a workload of $100\,000\,\mathrm{s}^{-1}$, the bolt becomes a bottleneck as indicated by its throughput of about $25\,000\,\mathrm{s}^{-1}$ that is smaller than the workload and spout throughput. Furthermore, for a workload of $1\,000\,000\,\mathrm{s}^{-1}$, the spout becomes a bottleneck; its throughput of about $120\,000\,\mathrm{s}^{-1}$ is smaller than the workload. For this case, the overall bottleneck that dictates the throughput of the topology is the bolt, and the overall topology throughput is limited to $25\,000\,\mathrm{s}^{-1}$. An interesting observation is that the spout has a larger capacity than the bolt. Because neither spout nor bolt do any computation, and because the bolt does not emit any output records, we derive that the cost for

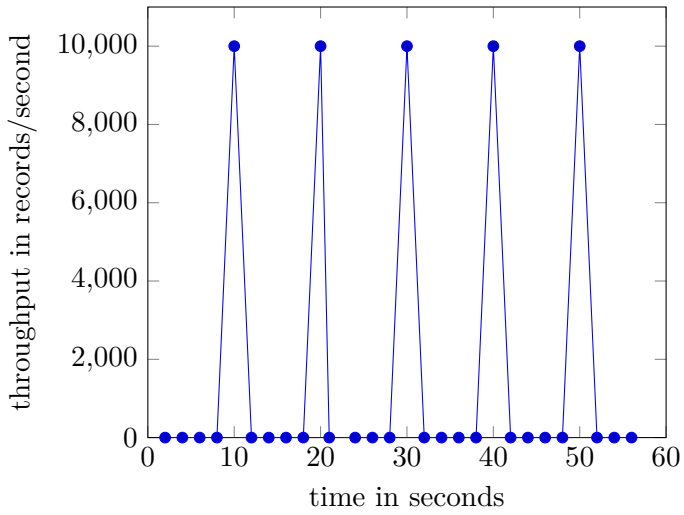

Figure 4.7: Bursty bolt throughput for spout output batch size $b_{\text{out}} = 10000$ and a workload of $1000\,\text{s}^{-1}$.

putting a record into the output queue (i. e., c_{emit}) and the cost for fetching a record from the input queue (i. e., c_{fetch}) are different.

We repeat the above experiment for the larger workloads of $100\,000\,\text{s}^{-1}$ and $1\,000\,000\,\text{s}^{-1}$, using spout output batch sizes of 10, 100, 1000, and 10 000 records. The result is shown in Figure 4.6. For an output batch size of 10 records, the bolt is no bottleneck for the smaller workload any longer. However, a batch size of 10 is not sufficient to eliminate bottlenecks for the larger workload, and both spout and bolt have a smaller throughput than the workload. Similarly, a batch size of 100 results in a spout and bolt bottleneck for a workload of $1\,000\,000\,\text{s}^{-1}$. A batch size $b_{\text{out}} = 1000$ eliminates the spout bottleneck, however, the bolt still has a smaller throughput (about $965\,000\,\text{s}^{-1}$) than the workload. Finally, the largest batch size of 10 000 records avoids all bottlenecks. Overall, the second part of the experiment shows that an increased output batch size not only increases the spout (i. e., producer) capacity, but also the downstream bolt (i. e., consumer) capacity.

We also evaluated the impact of large batch sizes for small workloads. Figure 4.7 shows the bolt throughput over time for a workload of $1000\,\text{s}^{-1}$ and a spout output batch size $b_{\text{out}} = 10000$ records. We observe that the data rate is bursty. Since the workload is small and the batch size is large, the spout emits batches in 10 second intervals. The bolt processes all records of an input batch quickly and stays idle until the next batch arrives. An interesting observation is that the bolt throughput spikes to $10\,000\,\text{s}^{-1}$, while the workload is only $1000\,\text{s}^{-1}$. Those large spikes are the result of the long idle times and the high processing capacity of the bolt. Furthermore, those spikes result in an increased processing latency that is dominated by batching, i. e., the time it takes to complete an output batch in the spout. Assuming a stable workload data rate of $1000\,\text{s}^{-1}$, the 95^{th} percentile of the processing latency is about 9.5 seconds. However, for a low workload batching is not required (c. f. Figure 4.5), and the data could be processed without a bottleneck and effectively zero latency. Therefore, the experiment shows that large batch sizes are undesired for

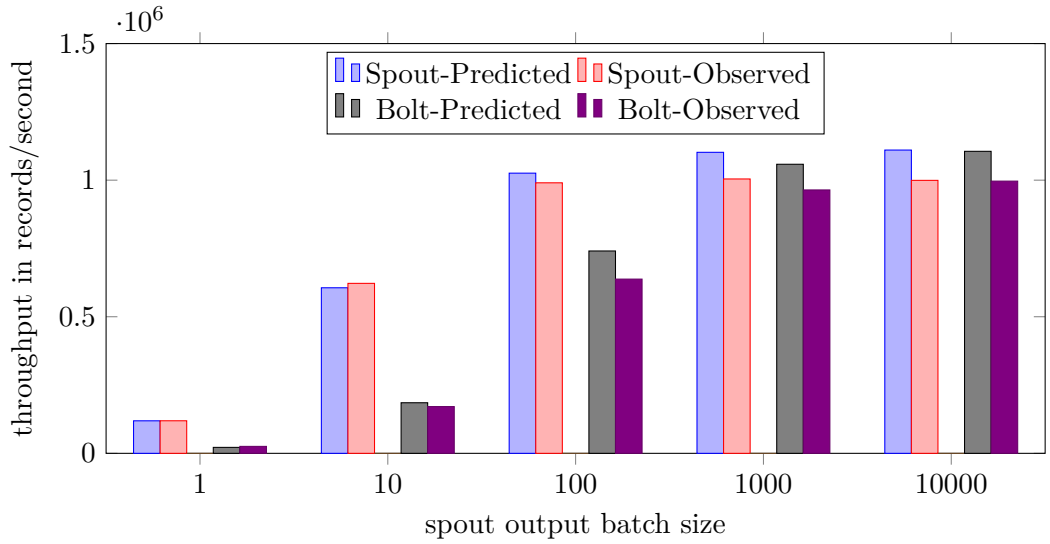

Figure 4.8: Predicted capacity and observerd throughput for different batch sizes and a workload of $1\,000\,000\,\mathrm{s}^{-1}$.

small workloads because they result in bursty data rates and increased processing latency.

To compare the observed throughput and the predictions of our cost model, we apply Equation 3.5 and Equation 3.8 (Section 3.2.1) to estimate the spout and bolt processing capacity. For the spout, there is no input stream and no input batch size. Furthermore, in our example topology the bolt is a sink and has no output stream and no output batch size. Thus, we remove the term $\frac{c_{\text{fetch}}}{b_{\text{in}}}$ from the equation for the spout, and we remove the term $\frac{c_{\text{emit}}}{b_{\text{out}}}$ from the equation for the bolt. We parametrize the cost model for the spout with $c_{\text{p}} = 900\,\text{ns}$ and $c_{\text{emit}} = 7500\,\text{ns}$, and for the bolt with $c_{\text{p}} = 900\,\text{ns}$ and $c_{\text{fetch}} = 45\,000\,\text{ns}$. We estimate those values using our experimental results from Figure 4.5 and Figure 4.6.

Figure 4.8 depicts the predicted capacity and observed throughput for the spout and the bolt based on Equation 3.5 and Equation 3.8. The predictions of our cost model are quite accurate and the average estimation error is about 8.65 % , with a minimum error of 0.11 % and a maximum error of 16.14 %.

Effective Input Batch Size

In the previous section, we evaluated the capacity prediction of our cost model with regards to batching. For this setup, we used a single producer-consumer pair, and modified the output batch size of the producer (which is the same as the input batch size of the consumer for this case). In general, the consumer input batch size depends on multiple factors like the output batch sizes and output data rates of its producers (Section 3.2.2). Each producer may have a different output batch size and different output data rate.

In this section, we evaluate the impact of multiple producers with different output batch sizes and different output data rates on the consumer capacity and throughput. We also compare the predictions of our cost model (Equation 3.21) to the

Table 4.1: Effective Input Batch Sizes Based on Equation 3.21

	b_{out}	1					5				
b_{out}	R_{out}	10	30	50	70	90	10	30	50	70	90
1	10					1					
	30				1						
	50			1							
	70		1								
	90	1									
5	10					3.6					5
	30				2.3					5	
	50			1.6					5		
	70		1.3					5			
	90	1.1					5				
10	10					5.3					9.1
	30				2.7					7.7	
	50			1.8					6.7		
	70		1.4					5.9			
	90	1.1					5.3				

observed consumer throughput. For this experiment, we use two spouts both connected to a single bolt. We vary the spout output data rates and output batch sizes while measuring the bolt throughput. We use a fixed data rate of $100\,000\,\text{s}^{-1}$ and distribute this data rate over both spouts. We start with an imbalanced workload and shift the load from one spout to the other spout until we reach a reversed imbalanced workload: in actual numbers, we use the following workload data rates: $10\,000\,\text{s}^{-1}/90\,000\,\text{s}^{-1}$, $30\,000\,\text{s}^{-1}/70\,000\,\text{s}^{-1}$, $50\,000\,\text{s}^{-1}/50\,000\,\text{s}^{-1}$, $70\,000\,\text{s}^{-1}/30\,000\,\text{s}^{-1}$, and $90\,000\,\text{s}^{-1}/10\,000\,\text{s}^{-1}$. The setup contains one combination that is a balance workload. We run the experiment for each output data rate combination with different output batch sizes for both spouts, in particular output batch sizes of 1/1, 1/5, 1/10, and 5/5.

Table 4.1 shows the predicted effective input batch sizes for the bolt (Equation 3.21) for all data rate and batch size combinations. For the cases with equal spout output batch sizes (e. g., 1/1 and 5/5) the effective bolt input batch size is the same as the spout output batch sizes, independent of the workload. However, different spout output batch sizes result in different effective bolt input batch sizes—even if the workload is balanced. This result shows that estimating the effective input batch size is rather difficult, and a manual configuration/optimization of batch sizes is hard in practice.

Similar to the previous experiments, both spouts generate random data in-memory (with a record size of 100 B). Figure 4.9 depicts the measured bolt throughput (we omit batch size combination 5/10 because the bolt is not a bottleneck for this case and its throughput is about $100\,000\,\text{s}^{-1}$ for all workloads). The figure shows that different batch sizes impact the bolt capacity dependent of their corresponding data rate. For example, using a batch size of 5 for the high throughput spout (i. e., 1/5 with workload $10\,000\,\text{s}^{-1}/90\,000\,\text{s}^{-1}$), increases the bolt capacity from about

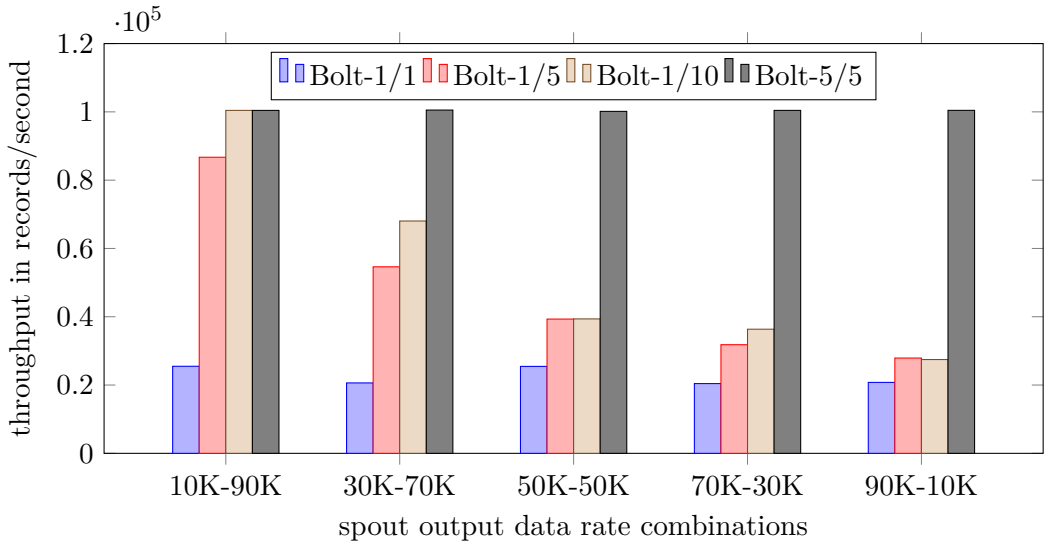

Figure 4.9: Bolt throughput for different combinations of spout output data rates and spout output batch sizes.

$25\,000\,\mathrm{s}^{-1}$ to $87\,000\,\mathrm{s}^{-1}$ (most left blue and red bars). The impact of an increased batch size is reduced for lower data rates: while the workload shifts from the second to the first spout the observed throughput is reduced, because the higher batch size of 5 is less efficient (compare red bar from left to right). Similarly, the bolt throughput decreases from left to right for a batch-size combination of 1/10. Using a batch size $b_{\text{out}} = 5$ for both spouts, the bolt is no bottleneck for all workloads. This result indicates that a batch size of 5 is sufficient to avoid a bolt bottleneck. An interesting observation is that there is no bottleneck for batch size combination 1/10 and workload $10\,000\,\mathrm{s}^{-1}/90\,000\,\mathrm{s}^{-1}$. This result aligns with the prediction of our cost model: Table 4.1 shows an effective batch size of 5.3 for this case that is larger than the required batch size of 5 to avoid a bottleneck. Overall, the experiment shows that we modeled the dependency of data rates and batch sizes for multiple producers accurately, and that our cost model computes a correct efficient input batch size.

4.3.2 Data Flow Optimization

In the previous section, we used micro-benchmarks and non-parallel execution graphs to evaluate the impact of batching on the task processing capacity. In this section, we use the Linear Road benchmark [ACG+04] to evaluate our cost model and resource minimization algorithm. We did not run the benchmark as proposed in the original paper, and we do not report a *L-factor* (a scalar value indicating the scaling factor, and hence maximum throughput of a system). Instead, we use the Linear Road data generator and a modified data flow program as depicted in Figure 4.10 in our evaluation. Our example data flow consists of a single spout that reads car position records line by line from a file. The spout sends the records to the *parse* operator that extracts the needed attributes like timestamp, vehicle id, vehicle speed, etc. Additionally, invalid records are filtered out by the parse operator. The parsed records are forwarded to the *agg* operator that computes the average speed over all

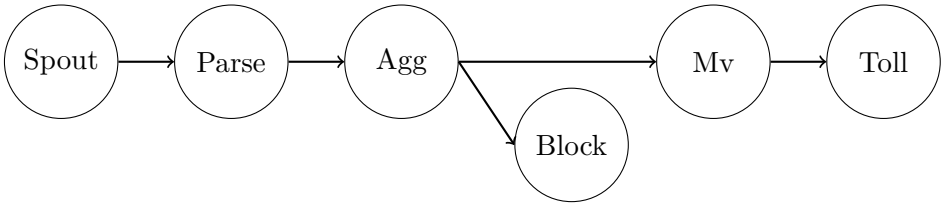

Figure 4.10: Modified Linear Road data flow program.

Table 4.2: Linear Road Meta Data

	Spout	Parse	Agg	Block	Move	Toll
c_{cpu} (in ns)	1000	3500	2600	5000	7000	4500
r_{out} (in bytes)	107	32	24	16	24	20
s	1.0	0.99	0.0003	0.002	1.0	1.0

Table 4.3: Optimized configuration w/ and w/o batching.

		Spout	Parse	Agg	Block	Mv	Toll	*RC*
w/o	*dop*	5	7	5	1	1	1	20
w/	*dop*	1	2	2	1	1	1	8
	b_{out}	15	387	1	$\perp$	1	$\perp$	

vehicles per road segment per minute. The agg operator forwards its output to two consumers: (1) the *block* operator, which computes the traffic capacity, and (2) the *mv* operator, which computes the average speed over a five minute hopping window with an advance of one minute. Finally, the *toll* operator computes the toll to be paid using the output of the mv operator. While the data is distributed randomly between spout and parse, all other connections use a hash-based partitioning on some record attributes.

In the following experiments, we used a processing cluster with similar configuration as described above. The main differences are that the nodes are connected with 1 Gbit Ethernet, and that we used Storm version 0.8.2. Additionally, we calibrated our cost model with ($c_{\text{fetch}} = 7400\,\text{ns}$ and $c_{\text{emit}} = 2600\,\text{ns}$). The cost factors and selectivity for each operator of our data flow program (Figure 4.10) is shown in Table 4.2.

Using a workload of $500\,000\,\text{s}^{-1}$, we computed two configurations for our data flow program—one with batching enabled, and one with batching disabled—to measure the impact of batching on the resource consumption. We modified Algorithm 3 accordingly such that it always uses a batch size of 1, and only minimized the *dop* for each operator. The result configurations for both cases are shown in Table 4.3. For the non-batching case, 20 tasks are deployed while for the batching case only 8 tasks are required. The result shows that batching increases task capacity and allows for a reduced resource consumption.

To evaluate if the computed configurations result in any bottleneck or overprovision the execution graph, we executed our data flow program with both configurations and measure the throughput for each node. Additionally, we manually

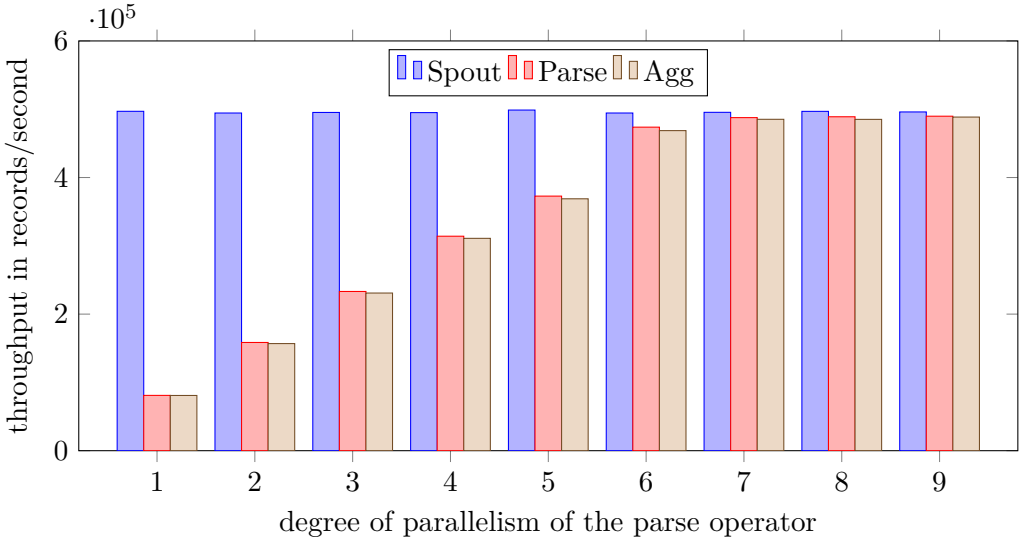

Figure 4.11: Operator throughput for different *dop* configurations of the parse operator with batching disabled.

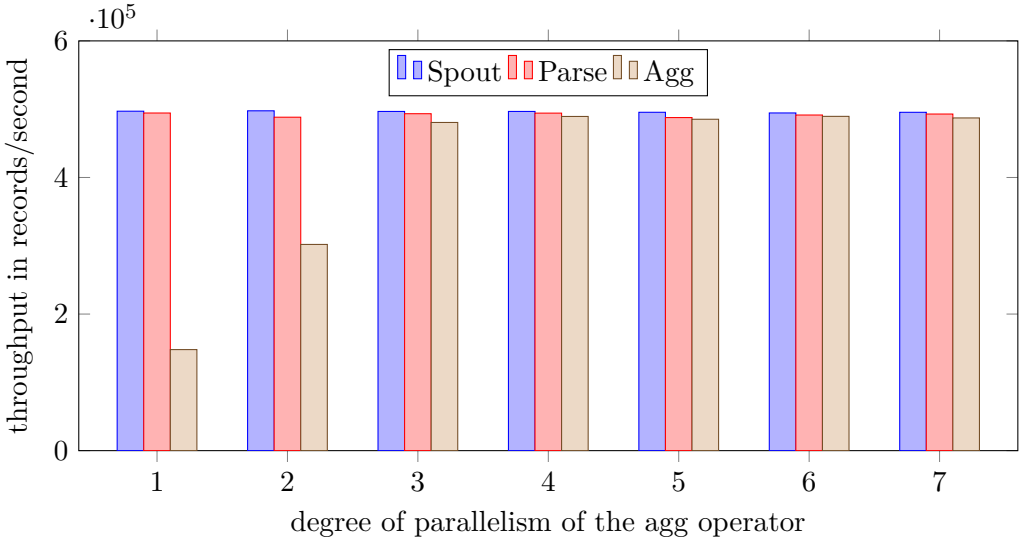

Figure 4.12: Operator throughput for different *dop* configurations of the agg operator with batching disabled.

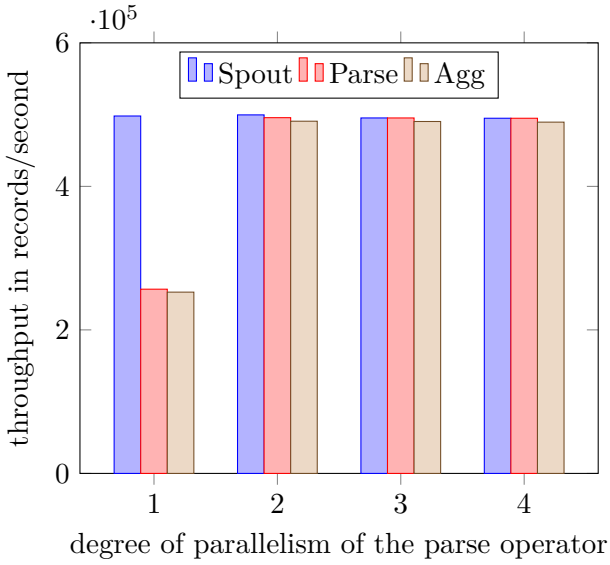

Figure 4.13: Operator throughput for different *dop* configurations of the parse operator with batching.

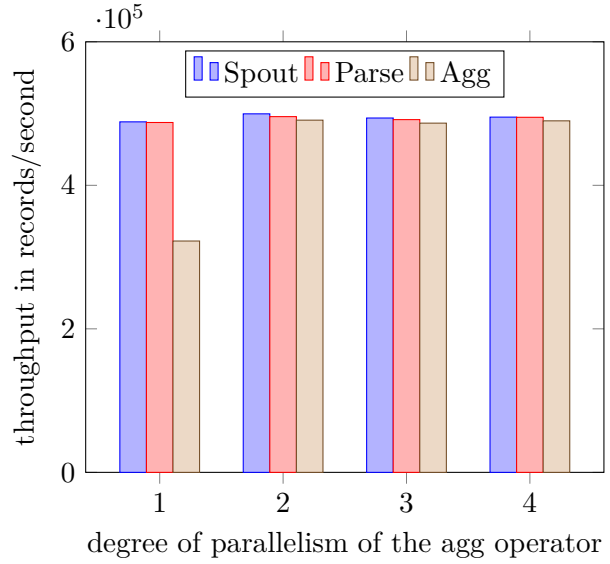

Figure 4.14: Operator throughput for different *dop* configurations of the agg operator with batching.

modified the *dop* to confirm if a lower *dop* results in a bottleneck, or if the computed *dop* over-provisions an operator. The results are depicted in Figure 4.11, Figure 4.12, Figure 4.13, and Figure 4.14. To vary the parallelism for each case we considered a range of $dop \in \{1, \ldots, dop^* + 2\}$, with dop^* being the optimized degree of parallelism from Table 4.3.

For the non-batching case, Figure 4.11 shows that the parse operator is a bottleneck if its *dop* is smaller than the computed $dop = 7$ from Table 4.3. The throughput of the agg operator grows with the throughput of the parse operator as its throughput is limited by the parse output data rate. Similarly, Figure 4.12 shows that the agg operator is a bottleneck for a $dop < 4$. For this case, the computed solution from Table 4.3 over-provisions the agg operator by one task.

Figure 4.13 and Figure 4.14 depict the corresponding results if we use batching. For both cases, the parse or agg operator is no bottleneck with a $dop = 2$ as computed by Algorithm 3.

Our results show that our cost model captures the runtime cost of data flow programs with regard to the operator parallelism and batch sizes accurately. Furthermore, Algorithm 3 minimizes the parallelism and computes corresponding batch sizes such that no bottleneck exists in the data flow.

4.4 Related Work

Aurora [CcC+02] has as similar overload concept to our model, with the main difference that Aurora is a centralized system and only considers CPU bottlenecks. Additionally, it introduces the notion of *headroom* that is the percentage of maximal available capacity for steady state (i. e., utilization in steady state). Headroom incorporates the fact that fluctuations in input data rates and data stream characteristics are expected to vary. While we described our optimization goal to fully utilize resources, we can easily adopt the headroom concept what should be beneficial for real-world deployments.

Bottleneck detection for parallel data flow programs is also proposed by Battré et al. [BHL+10] considering CPU and I/O (i. e., network). The main difference to our work is that they do not use a cost model but monitor a data flow program during execution. Bottlenecks are detected by leveraging observed operator state (i. e., PROCESSING vs. WAITING) and network channel state (i. e., READY vs. SATURATED). Furthermore, only parallelism is considered in their work, while we include operator output batch sizes.

The impact of batching on the processing latency in streaming data flows is studied by Lohrmann et al. [LWK12, LWK14, LJK15]. They propose dynamic batching, dynamic task chaining, and dynamic scaling based on measured latencies, with the goal to meet latency constraints. In our work, we use a best effort approach to reduce latency, without any quality of service guarantees. Furthermore, we determine the optimal parallelism and batch size analytically, rather than reactively. Our believe is that both approaches are complementary to each other and it is interesting future work to integrate them together.

Adaptive batching is also applied in Spark Streaming [DZSS14]. However, because Spark Streaming uses a micro-batching approach (Section 2.1) changing the batch size impacts the result and is not an internal and transparent optimization.

Similar to Lohrmann's work, dynamic scaling was also proposed for Storm [YM15, XPG16] and Heron [FAG+17]. For Storm, dynamic scaling is not done automatically, but must be triggered by the user, while Heron automatically adjusts the parallelism. However, both approaches focuses on parallelism and throughput only. For Heron, the authors also report that dynamic scaling may suffer from a cascading scaling effect, because upstream scale-out may lead to a downstream bottleneck. Hence, we propose to combine dynamic scaling with an analytical cost model approach, to predict downstream bottlenecks proactively and to avoid a cascading scaling effect.

Other work on elastic scaling focused on operator state handling [CFMKP13], minimizing latency spikes during a scaling event [HJHF14], or the integration of scaling and operator scheduling [FDM+17].

There is a large amount of work in the area of operator/task scheduling in distributed stream processing systems [WBH+08, BFc12, ABQ13, PHH+15, FDM+15]. The goal is to utilize cluster resources most efficiently, while avoiding the overload of individual servers. Our resource model is very simple and only based on tasks that are pinned to "task slots" (Section 4.2). A simple resource model is sufficient for our work. However, it is desirable to integrate our cost model with a scheduler using a more fine grained resource model.

4.5 Summary

In this section, we discussed the problem to identify bottlenecks in a data flow program and presented an algorithm that detects all bottlenecks in a data flow program given a data flow configuration. This algorithm is helpful if users specify a configuration manually. For the case that a bottleneck exists, we introduced an algorithm that estimates the achieved throughput. Second, we formally defined the optimization problem to minimize the used resources of a data flow deployment. We also introduced an algorithm that computes a solution to this optimization problem, i. e., our algorithm minimizes the degree of parallelism by computing corresponding batch sizes for each node in the data flow. The evaluation shows that batching increases task capacity and that our cost model and optimization algorithms are fairly accurate. We also discussed dynamic scaling and the shortcomings of such approaches. It is interesting future work to enhance dynamic scaling algorithms with an analytical cost model like ours.

Part III

Data Streaming Model

Chapter 5

The Dual Streaming Model

Contents

Even after many years of research, there is still no agreement in academia or industry on a standard stream processing model. The reason for this lack of standardization seems to lie in the semantic requirements that are highly application dependent. No existing model seems to satisfy a large enough range of use cases to be considered as a de-facto standard. At the same time, defining semantics for stream processing is demanding and existing proposals for data models and processing semantics span a wide range in the design space. One needs to cope with challenges imposed by the nature of data streams, the expressiveness of operators, and trade-offs faced when implementing a streaming model.

Data-related challenges: Data sources of contemporary streaming applications are inherently distributed (e. g., in IoT use cases) and commonly assign timestamps to streaming data, which induces a logical order. Yet, the physical order of data arriving at a stream processing system may be inconsistent with this logical order [SW04], due to imperfect clock synchronization, network delays, or sources buffering data while being disconnected for some time [ABC+15] (e. g., a mobile phone pushes data produced during a flight after reconnecting to the network). Hence, a stream processing model must be able to handle out-of-order data arrival.

Operator-related challenges: A stream processing model shall avoid implicit operator semantics, in order to achieve deterministic and well-defined processing results [BDD+10, DTM+13]. In particular, operators may be stateful and need to be based on the timestamps assigned to data in a stream, i. e., the logical order of streaming data [BBD+02]. Furthermore, the infinite nature of data streams implies a trade-off between processing cost, latency, and result completeness. Modeling this trade-off explicitly and in the light of operator properties (e. g., distributive, algebraic, and holistic functions [GCB+97]) allows users to reason about the system behavior.

System-related challenges: A central tenet of stream processing systems is online handling of data with low processing latency, which is of utmost importance. Hence, when implementing a stream processing model, one cannot rely on centralized algorithms in order to enable distributed evaluation of operators, that is required to provide low latency for high volume data streams. At the same time, data buffering (e. g., for reordering data to resolve inconsistencies of logical and physical ordering) [ACc+03b, TMSF03] shall be avoided, as it would compromise processing latency [KFD+10]. With records that are potentially delayed by hours (e. g., disconnected mobile phone during a flight) decoupling processing latency from handling out-of-order data is paramount.

Over the past decades, a plethora of stream processing models have been presented in the literature. Yet, we argue that existing models target only a subset of the aforementioned challenges. For instance, the seminal work on the Continuous Query Language (CQL) [ABW03] provides well-defined semantics for relational operators, but neglects issues stemming from out-of-order data. Other models, in turn, focus on handling of unordered data, but fail to address some of the other related challenges: stream punctuations [TMSF03] lead to increased processing latency; the "time-travel" mechanism of Borealis [AAB+05] stores the stream history, blurring the idea of online data handling; while existing models for update/delete data streams [BW01] lack formal semantics for operators or do not address the challenges of distributed processing [LTS+08, KFD+10].

In the following sections, we present the *Dual Streaming Model* to address the above questions and to unify existing approaches in a holistic model. The main idea of our model is to introduce state, captured by the relational notion of a table, as a first class citizen and to represent state as both a mutable collection of records and a stream of successive updates. This state representation induces a duality of tables and streams that enables reasoning over inconsistencies between the physical and logical order of streaming data.

5.1 Streams and Tables

In most existing streaming models, operators directly yield an output data stream and the treatment of operator state is considered to be an implementation detail. That is, models such as CQL [ABW03] ignore how inconsistencies of the physical and logical order of a stream may influence the computation of operator state and, therefore, the resulting output stream. Out-of-order records are neglected in the stream

Table 5.1: Formal Notation

Parameter	Abbr.	Notes
schemas	$\mathbb{R}$, $\mathbb{S}$, $\mathbb{T}$	for records, streams, tables
	$\mathbb{R} = \{T, K, V\}$ – timestamp, key, value $\mathbb{S} = \{O, T, K, V\}$ – offset, timestamp, key, value $\mathbb{T} = \{T, K, V\}$ – timestamp, key, value	
(table) record	$r = \langle t, k, v \rangle$	w/o offset
stream record	$r = \langle o, t, k, v \rangle$	w/ offset
record attributes	$r.o$, $r.t$, $r.k$, $r.v$	offset, timestamp, key, value
stream	$S[\mathbb{S}]$ / S	w/ and w/o explicit schema
table	$T[\mathbb{T}]$ / T	w/ and w/o explicit schema
table version	T_t	of version t
evolving table	$\vec{T}[\mathbb{T}]$ / $\vec{T}$	w/ and w/o explicit schema
record domains	$[\![\mathbb{R}]\!]$, $[\![\mathbb{S}]\!]$, $[\![\mathbb{T}]\!]$	set of all records w/ corresponding schema $\mathbb{R}$, $\mathbb{S}$, $\mathbb{T}$
stream domain	$S[\![\mathbb{S}]\!]$	set of all streams w/ schema $\mathbb{S}$
table domain	$T[\![\mathbb{T}]\!]$	set of all tables w/ schema $\mathbb{T}$
evolving table domain	$\vec{T}[\![\mathbb{T}]\!]$	set of all evolving tables w/ schema $\mathbb{T}$
modified key set	$\mathcal{K}_t$	set of modified keys of table T_t
set domain	$2^{\mathbb{D}}$	set of all sets of domain $\mathbb{D}$

processing model, so achieving consistent and correct results requires that evaluation is delayed until the logical order of records is established. As a consequence, the latency with which the output stream is computed grows linearly with the maximum unorder/delay of records [MLT+05, KFD+10], i. e., the difference between the timestamp of an out-of-order record r and the largest timestamp of all records with a smaller offset than r. Thus, latency is dominated by the characteristics of the data stream instead of the semantics and implementation of the operator.

To overcome this limitations, we propose a model in which the operator result is *updated continuously* [LWZ04, BGAH07]. Our model enables us to drop any assumptions about the consistency of the logical and physical order of records. Furthermore, the result of an operator may be viewed either statically, as its materialization from processing a stream up to an offset, or dynamically, as a stream of successive updates. Both views induce just two ways of programming with respect to *time*: one may process a stream of result updates or continuously query the materialized result to express the same computation.

The above idea is realized in our *Dual Streaming Model*, which comprises the notions of a *table*, a *table changelog stream*, and a *record stream*. Table 5.1 summarizes our notation of all terms that we define in this chapter or that we introduced in Section 2.4.

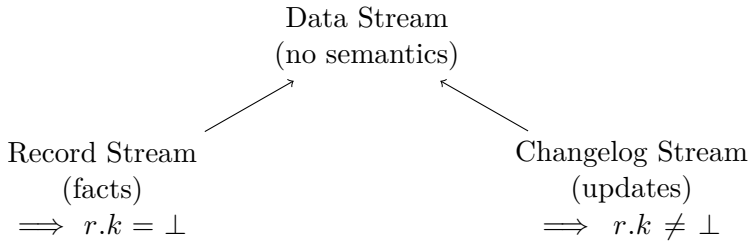

Figure 5.1: Data stream types and their relationship.

Definition 53 (Record Stream)**.** *A* record stream *is a data stream (Definition 10), with records representing* facts*. We also call a record stream a* fact stream*.*

Record streams are "regular" streams and used in most stream processing systems. In our model, keys are identifiers and thus, to represent immutable facts, it is required that each record in a record stream has a unique key. Instead of storing those unique identifiers explicitly we allow the identifier to be unassigned (i. e., $r.k = \bot$), with $\bot \neq \bot$, implying the same semantics. Hence, we refine the `app` operator from Definition 12 for records streams as follows:

Definition 54 (Record Stream Append)**.** *Given a record stream* $S[\mathbb{S}] = (r_0, \ldots, r_n)$ *and a record* r *with schema* $\bar{\mathbb{R}} = \mathbb{R} \backslash \{K\}$*,* $\texttt{app} : S[\![\mathbb{S}]\!] \times [\![\bar{\mathbb{R}}]\!] \to S[\![\mathbb{S}]\!]$ *appends* r *to* S*:*

$$\texttt{app}(S, r) = \texttt{app}(S, \langle t, \bot, v \rangle) \tag{5.1}$$

We use `app` as defined in Definition 12 on the right hand side of Equation 5.1. All variations of the append operator are denoted by `app` because it is clear from the context which one is used.

The second stream type in our model is a *changelog stream*.

Definition 55 (Changelog Stream)**.** *A* changelog stream *is a data stream (Definition 10), in which records represent* updates*. Updates are semantically based on record keys and timestamps, i. e., a later record with key* k semantically updates *an earlier record with the same key. Delete semantics are incorporated by interpreting records with* $\bot$*-value as deletes for earlier records with a corresponding key.*

It is important to note that there are no in-place updates or deletes in a changelog stream, because a changelog stream inherits all data stream properties and hence is append-only. Hence, a changelog stream records the complete history of all updates. Both record streams (Definition 53) and changelog streams (Definition 55) are data streams (Definition 10) with different semantic interpretation as depicted in Figure 5.1.

Finally, the result of an operator may be given by a *table* (Definition 11), which is updated for each processed input record. For instance, for an aggregation operator the grouping conditions (e. g., time windows or group-by clauses) define the primary key of the result table. A table models a snapshot of the result at a particular point in time. However, for stream processing operators that have temporal semantics, we need to reason about the content of a table over time. Hence, we consider tables to maintain multiple *versions* in parallel.

Definition 56 (Evolving Table)**.** *An* evolving table $\vec{T}$ *with schema* $\mathbb{T}$ *(denoted by* $\vec{T}[\mathbb{T}]$*) is a sequence of table versions, using timestamps as version numbers. The table version at timestamp* t *is denoted by* T_t *while an evolving table is denoted by* $\vec{T}[\mathbb{T}] = (T_0, \dots)$

In the remainder of this chapter, we use the terms *evolving table* and *table* interchangeably. While input data is processed, time progresses and a table evolves based on the timestamps of the updates: new table versions are added and older table versions might be updated if out-of-order data is processed. We define a generic *update* operator `update` that can be used to insert, modify, or delete records in a table. To define `update` correctly, we first introduce the notion of a *modified key set* for a table version.

Definition 57 (Modified Key Set)**.** *The* modified key set $\mathcal{K}_t$ *of a table* $T_t \in \vec{T}$ *with version* t *is the set of keys for which a modification with timestamp* t *was applied to* $\vec{T}$*.*

We use `insert` (Definition 19) and `delete` (Definition 20) as defined in Section 2.4.3 to define `update` as follows.

Definition 58 (Update Operator)**.** *Given a table* $\vec{T}$ *with schema* $\mathbb{T}$*, each table update is represented as a record* r *with schema* $\mathbb{R}$*. The update operator* $\texttt{update} : \vec{T}[\![\mathbb{T}]\!] \times [\![\mathbb{R}]\!] \to \vec{T}[\![\mathbb{T}]\!]$ *applies* r *to* $\vec{T}$ *by applying the update on all* $T_v \in \vec{T}$*:*

$$v = r.t \vee (v > r.t \wedge \forall v' : (r.t < v' \leq v \implies r.k \notin \mathcal{K}_v))$$

as

$$T_v = \begin{cases} T_v.\texttt{insert}(r) & \text{if } r.v \neq \bot \\ T_v.\texttt{delete}(r.k) & \text{otherwise} \end{cases} \tag{5.2}$$

and updating the modified key set of $T_{r.t}$*:*

$$\mathcal{K}_{r.t} = \mathcal{K}_{r.t} \cup \{r.k\} \tag{5.3}$$

If $T_{r.t}$ *does not exist in* $\vec{T} = (T_0, \dots, T_v)$ *(i. e.,* $r.t > v$*), new table versions are appended to* $\vec{T}$ *as*

$$\vec{T} = (T_0, \dots, T_v, T_{v+1}, \dots, T_{r.t})$$

with

$$T_v = T_{v+1} = \dots = T_{r.t}$$

and

$$\mathcal{K}_{v+1} = \dots = \mathcal{K}_{r.t} = \{\}$$

before the update is applied to $T_{r.t}$*.*

To reason about the state of an evolving table, we assign a *generation* number to it. Each applied update increments the generation by one. Thus, the table generation is the same as the offset of the latest processed input record. We denote a table $\vec{T}$ of generation g as $\vec{T}(g)$.

Example 14 (Evolving Table)**.** *Assume an empty table at generation zero* $\vec{T}(0) = (T_0)$ *with schema* $\{\texttt{key:String}, \texttt{value:Double}\}$*,* $T_0 = \{\}$*,* $\mathcal{K}_0 = \{\}$ *and the following sequence of updates:*

No.	t	k	v
0	*1*	A	*7.2*
1	*3*	A	*8.9*
2	*4*	B	*4.7*
3	*2*	C	*3.3*
4	*3*	A	$\perp$
5	*0*	B	*9.9*
6	*1*	A	*2.8*

Applying the updates evolves the table from generation 0 to 7, incorporating table versions $T_0, \ldots, T_4$ *(updates are highlighted):*

$\vec{T}(0)$*:*

$\mathcal{K}_0 = \{\}$

T_0

t	k	v

Initially, there is an empty table with version 0*.*

$\vec{T}(1) = \vec{T}(0).\texttt{update}(1, A, 7.2)$*:*

$\mathcal{K}_0 = \{\}$

T_0

t	k	v

$\mathcal{K}_1 = \{A\}$

T_1

t	k	v
1	A	*7.2*

The first update creates new table version T_1 *and inserts the new record into it. Additionally, the key* A *is added to* $\mathcal{K}_1$*.*

$\vec{T}(2) = \vec{T}(1).\texttt{update}(3, A, 8.9)$*:*

$\mathcal{K}_0 = \{\}$

T_0

t	k	v

$\mathcal{K}_1 = \{A\}$

T_1

t	k	v
1	A	*7.2*

$\mathcal{K}_2 = \{\}$

T_2

t	k	v
1	A	*7.2*

$\mathcal{K}_3 = \{A\}$

T_3

t	k	v
3	A	*8.9*

The second update creates two table versions, namely T_2 *and* T_3*, because the update timestamp is* 3*.* T_2 *is a copy of* T_1 *while* T_3 *contains the updated record for key* A*. It is important to note that* $\mathcal{K}_2$ *is empty, because neither the first nor the second update has timestamp* 2*.*

$\vec{T}(3) = \vec{T}(2).\texttt{update}(4, B, 4.7)$:

$\mathcal{K}_0 = \{\}$ T_0

t	k	v

$\mathcal{K}_1 = \{A\}$ T_1

t	k	v
1	A	7.2

$\mathcal{K}_2 = \{\}$ T_2

t	k	v
1	A	7.2

$\mathcal{K}_3 = \{A\}$ T_3

t	k	v
3	A	8.9

$\mathcal{K}_4 = \{B\}$ T_4

t	k	v
3	A	8.9
4	B	4.7

The third update inserts a record with new key B into the table. Hence, the newly create version T_4 contains two records. Since the record with key A was not updated at timestamp 4, $\mathcal{K}_4$ contains only B.

$\vec{T}(4) = \vec{T}(3).\texttt{update}(2, C, 3.3)$:

$\mathcal{K}_0 = \{\}$ T_0

t	k	v

$\mathcal{K}_1 = \{A\}$ T_1

t	k	v
1	A	7.2

$\mathcal{K}_2 = \{C\}$ T_2

t	k	v
1	A	7.2
2	C	3.3

$\mathcal{K}_3 = \{A\}$ T_3

t	k	v
3	A	8.9
2	C	3.3

$\mathcal{K}_4 = \{B\}$ T_4

t	k	v
3	A	8.9
4	B	4.7
2	C	3.3

The fourth update is the first out-of-order update that is applied to $\vec{T}$. It affects not only T_2, but also all future versions because none of them contain a record with key C. The actual condition if an update is propagated to future table versions does not depend on the existence of the key in the table, but the existence of the key in the modified key set. This condidition holds, as C is not in $\mathcal{K}_3$ or $\mathcal{K}_4$.

$\vec{T}(5) = \vec{T}(4).\texttt{update}(3, A, \perp)$:

$\mathcal{K}_0 = \{\}$ T_0

t	k	v

$\mathcal{K}_1 = \{A\}$ T_1

t	k	v
1	A	7.2

$\mathcal{K}_2 = \{C\}$ T_2

t	k	v
1	A	7.2
2	C	3.3

$\mathcal{K}_3 = \{A\}$ T_3

t	k	v
2	C	3.3

$\mathcal{K}_4 = \{B\}$ T_4

t	k	v
4	B	4.7
2	C	3.3

The fifth update is a delete that is also out-of-order. Hence, the record with key A is deleted from T_3 and T_4.

$\vec{T}(6) = \vec{T}(5).\texttt{update}(0, B, 9.9)$:

$\mathcal{K}_0 = \{B\}$ T_0			$\mathcal{K}_1 = \{A\}$ T_1			$\mathcal{K}_2 = \{C\}$ T_2			$\mathcal{K}_3 = \{A\}$ T_3			$\mathcal{K}_4 = \{B\}$ T_4		
t	k	v	t	k	v	t	k	v	t	k	v	t	k	v
			1	A	7.2	1	A	7.2						
0	B	9.9	0	B	9.9	0	B	9.9	0	B	9.9	4	B	4.7
						2	C	3.3	2	C	3.3	2	C	3.3

Similar to the fourth update, the sixth update is an out-of-order insert. However, it does not affect all *its future versions. Since a record with the same key B exist in T_4, T_4 is not modified.*

$\vec{T}(7) = \vec{T}(6).\texttt{update}(1, A, 2.8)$:

$\mathcal{K}_0 = \{B\}$ T_0			$\mathcal{K}_1 = \{A\}$ T_1			$\mathcal{K}_2 = \{C\}$ T_2			$\mathcal{K}_3 = \{A\}$ T_3			$\mathcal{K}_4 = \{B\}$ T_4		
t	k	v	t	k	v	t	k	v	t	k	v	t	k	v
			1	A	2.8	1	A	2.8						
0	B	9.9	0	B	9.9	0	B	9.9	0	B	9.9	4	B	4.7
						2	C	3.3	2	C	3.3	2	C	3.3

The last update to key A at timestamp 1 only affects T_1 and T_2. Even if T_3 does not contain a record with key A, A was explicitly deleted previously at timestamp 3 and thus is contained in $\mathcal{K}_3$, which implies that updating T_3 would be incorrect.

Our streaming model comprises of record streams, table changelog streams, and evolving tables. To this end, table changelog streams and evolving tables induce a duality between each other. Both represent the same data and can be transformed into each other without information loss. Consider the sequence of updates from Example 14. We can describe this sequence of updates as a changelog (Definition 55) and as shown in Example 14, this changelog can be transformed into a table by applying each record as an update to the table. At the same time, given any relational or evolving table, each modification to the table can be captured as an update record that is appended to a corresponding changelog stream. Hence, a changelog stream is a high level abstraction of a table write-ahead-log (WAL) as used in relational database systems. Figure 5.2 depicts the relationship between tables and changelog streams.

5.2 Stream Processing Operators

In this section, we introduce the stream processing operators of our model. We distinguish different types of operators depending on their properties. Operators may be stateless or stateful, have one or more input streams, and may be defined on special types of input streams only. Figure 5.3 summarizes all transformations between different types of streams/tables for different operators. Record streams are transformed into record streams (Section 5.2.1 and Section 5.2.3) with the exception of the aggregation operator that yields a result table (Section 5.2.2). Tables

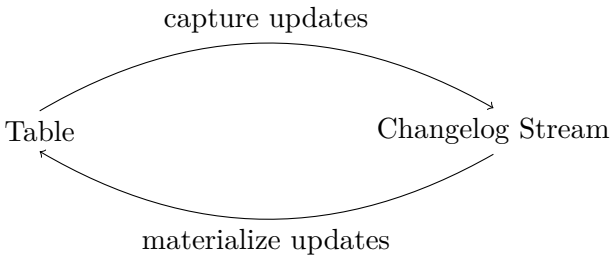

Figure 5.2: Duality of streams and tables.

are transformed into new tables (Section 5.2.4), unless joined with a stream (Section 5.2.3), in which case the result is an output stream. Furthermore, as discussed in Section 5.1, tables and table changelog streams can be converted into each other.

5.2.1 Record Stream Transformations

In the following, we define the operators of our streaming model in a declarative way that describes the expected output. To abstract over ordered and unordered input streams, we first define *stream equivalence* for record streams. Ordered streams are canonical representatives of an equivalence class of streams that contains both ordered and unordered streams. Hence, we define operators based on ordered input streams. Exploiting stream equivalence we consider an operator as *correct* if it computes a result on unordered input streams that is equivalent to the result as defined on ordered input streams.

Stream Equivalence and Operator Correctness

To define operator correctness we need to compare data streams. A straightforward comparison of two data streams leverages *stream equality* that is defined as follows:

Definition 59 (Record Stream Equality)**.** *Two record streams* $S_1[\mathbb{S}]$ *and* $S_2[\mathbb{S}]$ *are* equal*, denoted* $S_1 = S_2$*, iff:*

$$\forall r \in S_1 : (\exists \bar{r} \in S_2 : r = \bar{r}) \wedge \forall \bar{r} \in S_2 : (\exists r \in S_1 : \bar{r} = r) \tag{5.4}$$

Stream *equality* implies that two streams are *exactly* the same, i. e., all records appear in the exact same offset order. To define operator correctness, stream equality is too strict for two reasons:

Stream Comparison Let $S_1 = (r, s, t, v)$ be an ordered and $S_2 = (r, s, v, t)$ be an unordered input record stream. Both contain the same data, i. e., the same values with the same timestamps, but in different offset order. Since both streams contain the same data we want that an operator computes the same result for both input streams. However, stream *equality* cannot be used to express that both streams contain the same data. If two input streams are not "the same" (i. e., are not equal) it would be unreasonable to demand that an operator computes the same result for both inputs.

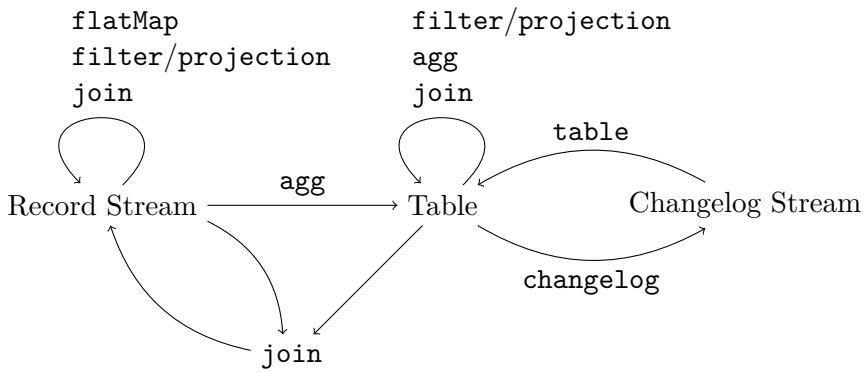

Figure 5.3: Transformations between record streams, changelog streams, and tables.

Similarly, we want to compare output streams. Assume we apply a `filter` operator to each stream yielding the following results:

$$\texttt{filter}(S_1) = \texttt{filter}((r, s, t, v)) = (r, t, v)$$

$$\texttt{filter}(S_2) = \texttt{filter}((r, s, v, t)) = (r, v, t)$$

Let us assume that (r, t, v) is a correct result, because it is computed based on an ordered input stream. Since both output streams contain the same data (i. e., timestamps and values), the result (r, v, t) should be considered as correct, even if it is not *equal* to the ordered result stream.

Thus, we need a different comparison criteria that allows us to compare two streams that contain the same data but are not equal. To compare record streams only timestamp and value should be considered. Offsets that describe the physical order should be ignored.

Infinite Input In practice, stream equality of output streams can never be guaranteed because input streams are potentially infinite and may have out-of-order data with arbitrary unorder/delay. Stream processing operators must be implemented using incremental algorithms to avoid blocking forever. However, operators could never emit any output if they need to guarantee that the output stream follows a certain offset order, because it is impossible to decide if there will be a future out-of-order record in the input stream.

Techniques like punctuations/watermarks [ABC$^+$15, TMSF03] try to address the out-of-order issue, by giving guarantees about input stream properties. However, as discussed in the beginning of this chapter, those approaches increase the processing latency and address the issue as a "preprocessing step", rather than as part of the processing model itself. Furthermore, in practice watermarks are often estimated and do not provide strict guarantees about out-of-order data.

To ignore offset order when comparing records streams we use *stream equivalence*. Since all records in a record stream have $k = \perp$, keys are ignored. We define stream equivalence based on multi-set equality [BGAH07, KS09] as follows:

Definition 60 (Record Stream Equivalence)**.** *Two record streams $S_1[\mathbb{S}]$ and $S_2[\mathbb{S}]$ are* equivalent*, denoted $S_1[\mathbb{S}] \equiv S_2[\mathbb{S}]$, iff their corresponding multi-sets of timestamp-value pairs are equal:*

$$S_1 \equiv S_2 \Leftrightarrow \texttt{MS}(\pi_{T,V}(S_1)) = \texttt{MS}(\pi_{T,V}(S_2)) \tag{5.5}$$

with `MS` *being an operator that transforms a sequence into a multi-set and π being a stream projection that returns the timestamp and value attribute for each record.*

The definition of record stream *equivalence* states that two record streams are equivalent, and thus belong to the same *equivalence class*, if they contain exactly the same records equally often but in potentially different offset order [BGAH07, KS09]. As record streams allow for duplicate records with same value and timestamp, we want to ensure that there is the same number of duplicates in both streams. Ignoring record offsets (i. e., physical order), we can interpret both record streams as multi-sets (also called bags) and state that two record streams are equivalent if their multi-sets are equal.

Based on record stream equivalence (Definition 60), we define correctness for record stream operators that output record streams as follows:

Definition 61 (Operator Correctness)**.** *Given two ordered data streams $O_1[\mathbb{S}_1]$ and $O_2[\mathbb{S}_2]$ and two unordered data streams $U_1[\mathbb{S}_1]$ and $U_2[\mathbb{S}_2]$. An unary stream processing operator $op : S[\![\mathbb{S}_1]\!] \to S[\![\mathbb{S}']\!]$ is correct, iff:*

$$O_1 \equiv U_1 \implies op(O_1) \equiv op(U_1) \tag{5.6}$$

A binary stream processing operator $op' : S[\![\mathbb{S}_1]\!] \times S[\![\mathbb{S}_2]\!] \to S[\![\mathbb{S}']\!]$ is correct, iff:

$$O_1 \equiv U_1 \wedge O_2 \equiv U_2 \implies op'(O_1, O_2) \equiv op'(U_1, U_2) \tag{5.7}$$

Definition 61 states that if two input streams are equivalent, both output streams must belong to the same equivalence class. Furthermore, ordered streams are canonical representatives for each equivalence class. Hence, defining operator semantics based on ordered input streams that yield ordered output streams is sufficient to define operator semantics for unordered input and output streams implicitly. It is important to note that Definition 61 can be extended to n-ary operators in a straightforward manner.

Stateless Operators

Given a record stream S, a stateless operator is applied to each record in the input stream in offset order and produces an order preserving output record stream. The output records inherit the timestamp from the input records. Examples of unary operators are filter, projection, or map that are all special cases of `flatMap`.

The *flatMap* operator `flatMap` is a second-order function that takes a record stream and a user-defined first-order function $f : [\![\mathbb{R}]\!] \to S[\![\mathbb{S}']\!]$ as parameters. For each record in input stream S, `flatMap` invokes f and appends all records from the result to the output stream. To define `flatMap`, we use `app` (Definition 54) as defined above as well as `fst` (Definition 14) and `sfx` (Definition 16) from Section 2.4.2.

Definition 62 (FlatMap Operator)**.** *Given a ordered record stream $S[\![\mathbb{S}]\!]$ and a function $f : [\![\mathbb{R}]\!] \to S[\![\mathbb{S}']\!]$ with stream-compatible schema $\mathbb{R}$. We define* $\texttt{flatMap}(S, f) : S[\![\mathbb{S}]\!] \times ([\![\mathbb{R}]\!] \to S[\![\mathbb{S}']\!]) \to S[\![\mathbb{S}']\!]$ *with $\mathbb{S}' = \{O, T, K', V'\}$ as follows.*

$$\texttt{flatMap}(S, f) = \begin{cases} () & \text{if } S = () \\ \texttt{app}(f(r), \texttt{flatMap}(\texttt{sfx}(S, 1), f)) & \text{otherwise} \end{cases} \tag{5.8}$$

with

$$r = \pi_{\mathbb{R}}(\texttt{fst}(S))$$

Two restrictions apply to f:

1. all output records from $f(r)$ must inherit the timestamp[1] of the input record, i. e., $\forall \bar{r} \in f(r) : \bar{r}.t = r.t$
2. the result for $f(r)$ must be finite

The second restriction is no restriction in practice, but must be part of the model: we demand that the computation of f is finite—otherwise, f would introduce a non-terminating operation. Thus, it would not be possible to process the input stream incrementally. Based on `flatMap` we define `map` and `filter` applying specific restrictions to the user-defined function f

- `map`: f returns exactly one record, i. e., $|f(r)| = 1$.
- `filter`: f returns either zero or one result record and does not modify the key or value i. e., $|f(r)| \leq 1 \wedge \forall \bar{r} \in f(r) : (\bar{r}.k = r.k \wedge \bar{r}.v = r.v)$

Expressing a filter operator as special case of `flatMap` allows us to reduce the number of operators in our model, however, it is rather unnatural: users would like to only specify a boolean predicate that returns `true` or `false` instead of a function f that can be provided to `flatMap`. Thus, we explicitly define the *filter* operator `filter` as second-order function that takes a user-defined filter predicate as input.

Definition 63 (Filter Operator)**.** *A* filter *operator is a second-order function* $\texttt{filter} : S[\![\mathbb{S}]\!] \times ([\![\mathbb{R}]\!] \to \{\bot, \top\}) \to S[\![\mathbb{S}]\!]$*. It takes an ordered data stream $S[\![\mathbb{S}]\!]$ and a user-defined filter predicate $p : [\![\mathbb{R}]\!] \to \{\bot, \top\}$ with stream-compatible schema $\mathbb{R}$ (Definition 8), and applies p to each record in the data stream. The result data stream contains all records for which p returns $\top$.*

$$\texttt{filter}(S, p) = \texttt{flatMap}(S, \mathit{flatMapFilter}(p)) \tag{5.9}$$

with

$$\mathit{flatMapFilter} : ([\![\mathbb{R}]\!] \to \{\bot, \top\}) \to ([\![\mathbb{R}]\!] \to S[\![\mathbb{S}]\!])$$

$$\mathit{flatMapFilter}(p) = r \to f(r)$$

$$\text{with } f(r) = \begin{cases} () & \text{if } p(r) = \bot \\ (\langle 0, r.t, r.k, r.v \rangle) & \text{if } p(r) = \top \end{cases}$$

[1] It would be possible to restrict f to not return a timestamp and let `flatMap` add the timestamp automatically. However, our definition simplifies the notation of Equation 5.8.

The helper function `flatMapFilter` takes a filter predicate and returns a function that can be provided to `flatMap`. Given an input record r, $f(r)$ returns an empty record stream if the filter predicate returns $\perp$. Otherwise, it returns a record stream with one record that carries the input record timestamp, key, and value. The offset is zero, because record streams are sequences that always start with offset zero. We emphasize that the created function obeys the restrictions of a filter function as described above.

A `projection` operator can be defined as a special case of `map` (similarly to `filter` that is a special case of `flatMap`). However, in our key-value model, `projection` is not useful and thus we omit it. For a generic tuple-based data model (similar to relational algebra), it is straightforward to define a projection operator on record streams.

Lemma 4. *The* `flatMap` *operator is correct (Definition 61), i. e., it computes the correct result on unordered input streams.*

Proof. The `flatMap` operator processes records independently of each other and thus computes the same result records independently of the record position in the input stream. Hence, if there are out-of-order records in the input stream, the result stream will contain the exact same records (only in different offset order) as if the input stream would not contain any out-of-order data. Therefore, the output stream of `flatMap` for an unordered input stream is equivalent to the output stream of an equivalent ordered input stream. □

Corollary 6. *All unary stateless record stream operators are correct.*

Proof. Since `flatMap` is the most generic unary stateless record stream operator, all other unary stateless stream operators can be expressed as `flatMap` with a corresponding user-defined function f. Since `flatMap` is correct, all other unary stateless record stream operators are correct. □

5.2.2 Record Stream Aggregation

The *aggregation* operator takes an input record stream and computes an aggregation result based on some grouping condition, similar to a `GROUP BY` clause in SQL. The aggregation returns a result table with the grouping attribute as primary key. Returning a table as the aggregation result differs from most stream processing systems that usually return a record stream instead.

The *aggregation* operator `agg` is a second-order function that takes a record stream, a user-defined first-order grouping function $g : [\![\mathbb{S}]\!] \to \mathbb{D}_{K'}$, and a user-defined first-order aggregation function $f : S[\![\mathbb{S}]\!] \to \mathbb{D}_{V'}$ as input, and produces a table as output. The input stream is split into substreams based on g, i. e., one substream per grouping attribute value. The aggregation function f is applied to each substream to compute the aggregation result.

Definition 64 (Aggregation Operator)**.** *Given an ordered record stream $S[\mathbb{S}]$, a function $g : [\![\mathbb{S}]\!] \to \mathbb{D}_{K'}$, and a function $f : S[\![\mathbb{S}]\!] \to \mathbb{D}_{V'}$. The* aggregation *operator*

$\texttt{agg} : S[\![\mathbb{S}]\!] \times ([\![\mathbb{S}]\!] \to \mathbb{D}_{K'}) \times (S[\![\mathbb{S}]\!] \to \mathbb{D}_{V'}) \to \vec{T}[\![\mathbb{T}]\!]$ *computes a table* $\vec{T}$ *with schema* $\mathbb{T}$ = *{T, K',V'} as:*

$$\texttt{agg}(S, g, f) = \vec{T} = (T_0, \cdots, T_{\hat{t}}) \tag{5.10}$$

with

$$\hat{t} = \max\{r.t | r \in S\}$$

and

$$\begin{aligned}\forall T_t \in \vec{T} : (S_t = S.\texttt{filter}(r \to r.t \leq t) \ \wedge \\ K = \{g(r) | r \in S_t\} \ \wedge \\ |T_t| = |K| \ \wedge \\ \forall k \in K : (S_t^k = S_t.\texttt{filter}(r \to g(r) = k) \ \wedge \\ \hat{t}_k = \max\{r.t | r \in S_t^k\} \ \wedge \\ \langle \hat{t}_k, k, f(S_t^k) \rangle \in T_t) \ )\end{aligned}$$

The result of a record stream aggregation is a table that grows without bound, since new table versions are added while time progresses. In practice, applications are usually only interested in the latest result and hence it it sufficient to only store the most recent T_t.

Example 15 (Record Stream Aggregation)**.** *Assume an ordered record stream S that represent web-site clicks. Let the web-site click stream contain information about the country of origin and we want to compute the click count per country. The timestamp/value pairs of the click stream are as follows:*

$$S = (\langle 0, GER \rangle, \langle 1, GER \rangle, \langle 1, US \rangle, \langle 2, US \rangle)$$

The corresponding aggregation result table is:

$\mathcal{K}_0 = \{GER\}$ T_0

t	k	v
0	GER	1

$\mathcal{K}_1 = \{GER, US\}$ T_1

t	k	v
1	GER	2
1	US	1

$\mathcal{K}_2 = \{US\}$ T_2

t	k	v
1	GER	2
2	US	2

The record stream aggregation operator as defined in Equation 5.10 applies the aggregation function f to the substream S_t^k. If f is a holistic aggregation functions [GCB+97] it would be required to store S_t^k. However, S_t^k may grow without bound over time and therefore we do not allow holistic aggregation functions for non-windowed aggregations.[2] However, we allow algebraic aggregation functions [GCB+97] if their underlying functions are commutative and associative. For this case, a new table version T_{t+1} can be computed incrementally based on T_t and all input stream records with timestamp $t + 1$.

We introduce Algorithm 4 that computes `agg` incrementally, given a *commutative and associative* aggregation function $f : [\![\mathbb{R}]\!] \times [\![\mathbb{T}]\!] \to \mathbb{D}_{V'}$. To describe the incremental algorithm that computes `agg` for ordered[3] input streams, we use `update` (Definition 58) as defined above as well as `lookup` as defined next.

[2] We discuss windowed aggregations later.

[3] We discuss unordered input streams later.

Algorithm 4: Incremental Record Stream Aggregation

Input: $S[\mathbb{S}]$; $g : [\![\mathbb{S}]\!] \to \mathbb{D}_{K'}$; $f : [\![\mathbb{S}]\!] \times ([\![\mathbb{T}]\!] \cup \{\bot\}) \to \mathbb{D}_{V'}$ with $\mathbb{T} = \{T, K', V'\}$
Output: continuously updated table $\vec{T}[\mathbb{T}]$

$\vec{T} \leftarrow (\{\})$ `// init result with empty table`
foreach $r \in S$ **do** `// never terminates`
  $k \leftarrow g(r)$
  $\bar{v} \leftarrow \vec{T}.\texttt{lookup}(k)$
  $v' \leftarrow f(r, \bar{v})$
  $\vec{T}.\texttt{update}(r.t, k, v')$

Definition 65 (Lookup Operator)**.** *Given a table $\vec{T} = (T_0, \ldots, T_t)$ with schema $\mathbb{T} = \{T, K, V\}$—that contains t table versions—and a key $k \in \mathbb{D}_K$. The* lookup *operator* `lookup` $: \vec{T}[\![\mathbb{T}]\!] \times \mathbb{D}_K \to \mathbb{D}_V$ *returns the latest value $v \in \mathbb{D}_V$ of k in $\vec{T}$:*

$$v = T_t.\texttt{lookup}(k) \tag{5.11}$$

Definition 65 uses `lookup` (Definition 18) to get the value of the key from the latest table version. Because `lookup` may return $\bot$ if the key does not exist, the aggregation functions f must accept $\bot$ as input.

Algorithm 4 computes a record stream aggregation over an ordered input record stream. It processes the input records one-by-one and incrementally updates the aggregation result based on the currently processed record. First, `agg` calls the grouping function g with the current record r to determine the grouping attribute-value $k = g(r)$ that is the primary key of the corresponding aggregation result in the result table. Based on k, the current aggregation result $\bar{v}$ is retrieved from the table. Afterwards, the commutative and associate aggregation function f is applied to the current record and the current aggregation result. Finally, the result table is updated with the new aggregation result returned by f, before the next record is processed.

Lemma 5. *Algorithm 4 computes the correct result based on Definition 64.*

Proof. Let r be the currently processed record and let $\vec{T} = (T_0, \ldots, T_v)$ be the current result table.

(1) *The latest table version in $\vec{T}$ is of timestamp $v \leq r.t$*: Because the result table is initialzed as an empty table, and because the input stream S is ordered, `update` (Definition 58) either modifies the latest existing table version, or it creates a new table version for timestamp $r.t$. Hence, it holds that $v \leq r.t$ for each processed record.

Assuming that the table is updated correctly for each input record (as proven in (3) below), it holds that:

(2) *The current aggregation result of $g(r)$, i. e., the aggregation result of $g(r)$ at timestamp $r.t$ before r is processed, is stored in the latest table version T_v*: Because of (1), $r.t \not< v$. If $r.t = v$, it's clear that (2) holds because S is ordered. If $r.t > v$, it holds that all not-yet existing table versions T_x with $v < x \leq r.t$ are equal to T_v; S is ordered and therefore (a) all updates before $r.t$ are reflected in T_v already, (b) no

update exists with timestamp between v and $r.t$ (both bounds exclusive), and (c) r is the first update at timestamp $r.t$ (otherwise, $r.t$ would not be larger than v based on (1)). Hence, getting the current aggregation result from T_v is correct for $r.t > v$.

(3) *The table is updated correctly for each processed record*: Because of (2) and because f is commutative and associative, the computed new aggregation result is equal as the aggregation result from Definition 64 with regard to $\texttt{pre}(S, r.o + 1)$ (Definition 15). Therefore, the table is updated correctly for each processed record.

There is no cyclic dependency between (2) and (3) because the result is initialized with the empty table, and hence, the assumption that (3) holds is not required for the first processed record to prove (2). Additionally, when r is processed (3) holds for all previously processed records. Because (3) holds for each record, Lemma 5 is correct. □

Windowed Aggregation

Aggregations in stream processing are usually based on a grouping attribute (similar to a `GROUP BY` clause in the relational model) and *windows* [BBD+02]. Windows are additional grouping conditions. In contrast to a `GROUP BY` condition that generates "vertical" substreams, windows split the input stream "horizontal" usually based on timestamps.[4] Our model supports windowed aggregations by encoding the grouping attribute and a window-ID [LMT+05] in the primary key of the result table. The table stores the current aggregate per key and per window. Hence, we do not distinguish between windowed and non-windowed aggregations explicitly. However, a windowed aggregation applies the aggregation function to finite windows[5], and thus, holistic aggregation functions are permitted for this case if they are commutative and associative. Since record timestamps are not guaranteed to be unique, there may be multiple ordered record streams in the same equivalence class. The aggregation result over all those ordered record streams is only the same if the aggregation function is commutative and associative

The two most common fixed size time windows are non-overlapping tumbling windows and overlapping hopping windows. A tumbling window has a single parameter ω that defines the size of the window in time units. Windows are aligned to the time epoch, with window start timestamp being inclusive and window end timestamp being exclusive. For example, a tumbling window with a size of 10 time units, has the following window boundaries:

$$[0, 10), [10, 20), [20, 30), \ldots, [n \cdot \omega, n \cdot \omega + \omega)$$

Hopping windows have a second parameter δ that defines the advance of the window. For example, a hopping window with a size of 10 time units and an advance of 5 time units, has the following window boundaries:

$$[0, 10), [5, 15), [10, 20), [15, 25), [20, 30), \ldots, [n \cdot \delta, n \cdot \delta + \omega)$$

[4] Count-based windows are also common, however, they are inherently non-deterministic if record order is defined on event-time, and thus, we omit count-based windows.

[5] This holds for fixed size time windows—for session windows [ABC+15], it is conceptually possible that a window grows unbounded. In practice, session windows are bounded and thus holistic aggregation functions may be applied.

To this end, tumbling windows are a special case of hopping windows with $\delta = \omega$. For both window types, the grouping attribute (i. e., primary key of the result table) is a pair $\langle k, w\text{-}Id \rangle$, with k being a value-extracted grouping attribute and $w\text{-}Id$ being a unique window-ID. For fixed size time windows, $w\text{-}Id$ can be expressed as the window start timestamp.

For tumbling windows, each record is contained in exactly one window with window start timestamp:

$$w\text{-}Id(r) = \left\lfloor \frac{r.t}{\omega} \right\rfloor \cdot \omega$$

For hopping windows, records may be contained in multiple windows. The window start timestamps are computed as follows:

$$w\text{-}Id(r) = \begin{Bmatrix} \left\lfloor \frac{r.t - \omega}{\delta} \right\rfloor \cdot \delta + 1 \cdot \delta, \\ \left\lfloor \frac{r.t - \omega}{\delta} \right\rfloor \cdot \delta + 2 \cdot \delta, \\ \dots, \\ \left\lfloor \frac{r.t - \omega}{\delta} \right\rfloor \cdot \delta + \#w \cdot \delta \end{Bmatrix}$$

with $\#w(r) = \lfloor \frac{r.t}{\delta} \rfloor - \lfloor \frac{r.t-\omega}{\delta} \rfloor$ being the number of windows that contain a specific record. It is important to note that for hopping windows, not all records may be contained in the same number of windows. Hence, $\#w$ must be computed on a per record basis.

Example 16 (Hopping Window)**.** *Assume a hopping window with window size* $\omega = 10$ *and advance* $\delta = 3$*. The following window boundaries apply (we use* inclusive *upper bounds to only show timestamps that belong to a window):*

$$\begin{aligned}
\text{w-Id}_0 &= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] \\
\text{w-Id}_3 &= \quad [3, 4, 5, 6, 7, 8, 9, 10, 11, 12] \\
\text{w-Id}_6 &= \quad\quad [6, 7, 8, 9, 10, 11, 12, 13, 14, 15] \\
\text{w-Id}_9 &= \quad\quad\quad [9, 10, 11, 12, 13, 14, 15, 16, 17, 18] \\
\text{w-Id}_{12} &= \quad\quad\quad\quad [12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
\end{aligned}$$

The window boundaries show that records fall either into 3 *or* 4 *windows. For example, records with timestamp* 9 *or* 12 *belong to* 4 *windows, while records with timestamp* 10 *or* 11 *belong to only* 3 *windows.*

However, in Definition 64 the grouping function g returns a single grouping attribute and hence overlapping windows are not supported. To allow for overlapping windows, we change g to return a set of grouping keys instead: $g : [\![\mathbb{S}]\!] \to 2^{\mathbb{D}_{K'}}$. Furthermore, we modify Definition 64 to compute the set of primary keys as:

$$K = \{k | r \in S_t \wedge k \in g(r)\}$$

and, the corresponding substreams as:

$$S_t^k = S_t.\texttt{filter}(r \to k \in g(r))$$

Example 17 (Windowed Aggregation)**.** *Assume the same ordered input stream as from Example 15:*

$$S = (\langle 0, GER\rangle, \langle 1, GER\rangle, \langle 1, US\rangle, \langle 2, US\rangle)$$

Instead of counting all clicks per country, we count clicks using a tumbling window with $\omega = 2$*. The corresponding aggregation result table is:*

$\mathcal{K}_0 = \{\langle GER, 0\rangle\}$ $\mathcal{K}_1 = \{\langle GER, 0\rangle, \langle US, 0\rangle\}$ $\mathcal{K}_2 = \{\langle US, 2\rangle\}$

T_0

t	k	v
0	*GER,0*	*1*

T_1

t	k	v
1	*GER,0*	*2*
1	*US,0*	*1*

T_2

t	k	v
1	*GER,0*	*2*
1	*US,0*	*1*
2	*US,2*	*1*

The windows have boundaries $\text{w-Id}_0 = [0, 2)$ *and* $\text{w-Id}_2 = [2, 4)$*. The first window is instantiated twice, once for grouping attribute GER and once for US. The second window is instantiated only for US. Hence, there are three primary keys* $\langle GER, 0\rangle$*,* $\langle US, 0\rangle$*, and* $\langle US, 2\rangle$ *in the final result (*T_2*). Both records with value GER belong to* w-Id_0 *and thus the count is* 2*. The two "US" records belong to separate windows and thus the count is* 1 *each.*

For windowed aggregation, the result table grows without bound in two dimensions: (1) similar to the non-windowed case, new table versions are added over time; (2) additionally, the size of individual table versions T_t grows unbounded, because with advancing time, the set of primary keys grows without bound due to a unbounded number of created windows. We discuss this issue in more detail in Section 5.3.

Aggregation of Unordered Record Streams

Definition 64 defines the expected result of the record stream aggregation operator based on ordered input streams. Because the result of an aggregation is a table, we use table equality to define result correctness for unordered input streams.

Definition 66 (Operator Correctness)**.** *Given an ordered data stream* $O[\mathbb{S}]$ *and a unordered data stream* $U[\mathbb{S}]$*. The* `agg` *operator is correct, iff:*

$$O \equiv U \implies \texttt{agg}(O, g, f) = \texttt{agg}(U, g, f) \tag{5.12}$$

In contrast to Definition 61 we demand that the result tables are *equal*, if the corresponding input record streams are *equivalent* (Definition 60). Hence, even if there is out-of-order data in the input stream, the aggregation must compute the exact same result table. If out-of-order data is in the input data stream, the result table may have a different intermediate result though: if the stream prefixes are not equivalent, i. e., $\texttt{pre}(O, n) \not\equiv \texttt{pre}(U, n)$ the corresponding intermediate result tables may be different [BGAH07].

Example 18 (Out-of-Order Aggregation)**.** *Let S' be the following data stream that is equivalent to S from Example 15:*

$$S' = (\langle 0, GER\rangle, \langle 1, GER\rangle, \langle 2, US\rangle, \langle 1, US\rangle)$$

Processing S and S', evolves the output table from generation 0 to generation 3. The first 2 records of both data streams are the same and hence, the first two table generations are the same. However, $\vec{T}(3) \neq \vec{T}'(3)$:

$\vec{T}(3) = \texttt{agg}((\langle 0, GER\rangle, \langle 1, GER\rangle, \langle 1, US\rangle), \dots)$*:*

$\mathcal{K}_0 = \{GER\}$ T_0

t	k	v
0	*GER*	*1*

$\mathcal{K}_1 = \{GER, US\}$ T_1

t	k	v
1	*GER*	*2*
1	*US*	*1*

$\vec{T}'(3) = \texttt{agg}((\langle 0, GER\rangle, \langle 1, GER\rangle, \langle 2, US\rangle), \dots)$*:*

$\mathcal{K}_0 = \{GER\}$ T_0

t	k	v
0	*GER*	*1*

$\mathcal{K}_1 = \{GER\}$ T_1

t	k	v
1	*GER*	*2*

$\mathcal{K}_2 = \{US\}$ T_2

t	k	v
1	*GER*	*2*
2	*US*	*1*

Lemma 6. *Both intermediate results from Example 18, i. e., $\vec{T}(3)$ and $\vec{T}'(3)$, are correct.*

Proof. The fact that $\vec{T}(3) \neq \vec{T}'(3)$, does not violate Definition 66 because:

$$\texttt{pre}(S, 3) \not\equiv \texttt{pre}(S', 3)$$
$$(\langle 0, GER\rangle, \langle 1, GER\rangle, \langle 1, US\rangle) \not\equiv (\langle 0, GER\rangle, \langle 1, GER\rangle, \langle 2, US\rangle)$$

Because the processed input streams are not equivalent after processing 3 records each, it is not required that the result tables are equal. Furthermore, both results $\vec{T}(3)$ and $\vec{T}'(3)$ are computed for a prefix that does not contain out-of-order data. Hence, both results (in particular $\vec{T}'(3)$) are correct based on Definition 64. □

In Example 18, $\texttt{pre}(S, 1) \equiv \texttt{pre}(S', 1)$, $\texttt{pre}(S, 2) \equiv \texttt{pre}(S', 2)$, and $\texttt{pre}(S, 4) \equiv \texttt{pre}(S', 4)$. Hence, it is required that $\vec{T}(1) = \vec{T}'(1)$ $\vec{T}(2) = \vec{T}'(2)$, and $\vec{T}(4) = \vec{T}'(4)$.

The incremental aggregation computation as introduced in Algorithm 4, expects ordered input streams, and may compute an incorrect result for unordered input streams (incorrect result highlighted in red):

Algorithm 5: Out-of-Order Record Stream Aggregation

Input: $S[\mathbb{S}]$; $g : [\![\mathbb{S}]\!] \to 2^{\mathbb{D}_{K'}}$; $f : [\![\mathbb{S}]\!] \times ([\![\mathbb{T}]\!] \cup \{\bot\}) \to \mathbb{D}_{V'}$
with $\mathbb{T} = \{T, K', V'\}$
Output: continuously updated table $\vec{T}[\mathbb{T}]$

$\vec{T} \leftarrow (\{\})$ `// init result with empty table`
foreach $r \in S$ **do** `// never terminates`
  if $r.t > \vec{T}.t$ **then** `// add new table version`
    foreach $k \in g(r)$ **do**
      $\bar{v} \leftarrow \vec{T}.\texttt{lookup}(k)$
      $v' \leftarrow f(r, \bar{v})$
      $\vec{T}.\texttt{update}(r.t, k, v')$
  else
    $T^* = \{T_x \in \vec{T} | x = r.t \vee (x > r.t \wedge g(r) \cap \mathcal{K}_x \neq \emptyset)\}$
    foreach $T_x \in T^*$ **do**
      if $x = r.t$ **then** `// oldest table version to be updated`
        $K \leftarrow g(r)$
      else
        $K \leftarrow \{k \in g(r) | k \in \mathcal{K}_x\}$
      foreach $k \in K$ **do**
        $\bar{v} \leftarrow T_x.\texttt{lookup}(k)$
        $v' \leftarrow f(r, \bar{v})$
        $\vec{T}.\texttt{update}(x, k, v')$

$\vec{T}'(4) = \texttt{agg}((\langle 0, GER\rangle, \langle 1, GER\rangle, \langle 2, US\rangle, \langle 1, US\rangle), \dots)$:

$\mathcal{K}_0 = \{GER\}$ T_0

t	k	v
0	GER	1

$\mathcal{K}_1 = \{US, GER\}$ T_1

t	k	v
1	GER	2
1	US	1

$\mathcal{K}_2 = \{US\}$ T_2

t	k	v
1	GER	2
2	US	1

Since the input stream is assumed to be ordered, Algorithm 4 updates only a single table version corresponding to the record timestamp, i. e., T_1 gets a new count of 1 for the US. However, to handle out-of-order data correctly, it is required to additionally update all newer table versions that exist in $\vec{T}$, i. e., T_2 should have been updated to a count of 2 for the US. To address this issue, we introduce an incremental aggregation algorithm (Algorithm 5) for unordered record streams. Algorithm 5 uses the generic grouping function that returns a set of grouping keys and hence supports windowed-aggregations. To describe the algorithm, we use `update` (Definition 58) and evolving table `lookup` (Definition 65) from above as well as table version `lookup` (Definition 18) from Section 2.4.2.

Algorithm 5 takes a record stream $S[\mathbb{S}]$ that may contain out-of-order records, a grouping function $g : [\![\mathbb{S}]\!] \to 2^{\mathbb{D}_{K'}}$, and a commutative and associative aggregation function $f : [\![\mathbb{S}]\!] \times [\![\mathbb{T}]\!] \to \mathbb{D}_{V'}$ with $\mathbb{T} = \{T, K', V'\}$ as input. The algorithm processes

S incrementally and updates the aggregation result table for each processed input record. There are two main cases: first, the processed record might advance time and a new table version is added to the result (Lines 6-10). For this case, the table is updated for each grouping attribute-value as returned by $g(r)$. The update for a single key, is the same as in Algorithm 4: the algorithm first receives the old aggregation value $\bar{v}$ from the table, computes a new aggregation result using f, and updates the result table accordingly. Second, the processed record updates an existing table version (Lines 12-21). For this case, the table version that matches the record timestamp, as well as all newer table versions that contain at least one grouping attribute-value in their modified key set, are updated explicitly. Those table versions are computed in Line 12. To update those table versions, the table version that matches the record timestamp is updated for all grouping attribute-values (Line 15), which may insert new keys into the table. All other tables only need an *explicit* update for all grouping attribute-values that are contained in the corresponding modified key set (Line 17). All grouping attribute-values that are not contained in the modified key set are updated *implicitly* when an older table version is updated (c. f. Definition 58). The actual table update per key (Lines 19-21) is similar to Lines 8-10, however, the current aggregation value $\bar{v}$ is retrieved from the corresponding table version (Line 19) instead.

Lemma 7. . *Algorithm 5 computes the correct aggregation result (Definition 64 and Definition 66) for unordered input streams.*

Proof. Algorithm 5 handles two different cases: (1) A new table version is added: For this case, we apply Lemma 5, because the update of each table key is independent from the updates to all other table keys. (2) An existing table version is updated: For this case, all table versions beginning with the table version corresponding to the record timestamp must be updated for all grouping attribute-values. For the table version that corresponds to the record timestamp, this is guaranteed because the set of all keys K is set to $g(r)$. Furthermore, based on Definition 58, each explicit update implicitly updates all existing newer table versions that do not contain the corresponding key in their modified key set. Updating newer versions implicitly is correct because it ensures that all results that are equal, i. e., not modified in a newer table version, are updated together. If a table version has a key k in its modified key set, it implies that the aggregation result for k is different to its previous version. Therefore, it needs to be explicitly updated with the new record. Since Algorithm 5 updates exactly those table versions for exactly those keys, the result table is correctly updated for each processed record. □

5.2.3 Record Stream Joins

We include a variety of equi-join operators for streams and tables in our model, i. e., stream-stream, stream-table, and table-table joins. In this section, we introduce stream-stream and stream-table joins, while we discuss table-table joins in Section 5.2.4.

All joins require an equi-join condition to allow for a distributed join computation. As discussed in the beginning of this chapter, a distributed operator implementation is desired in modern large-scale stream processing systems: requiring an equi-join condition allows for a data-parallel join computation (Section 2.2.1). Both

inputs can be split into N shards, using for example hash- or range-partitioning (Section 2.2.2) on the join-attribute. Sharding the input allows us to compute the join result using N machines in parallel by co-partitioning the corresponding shards on the same machine.

Stream-Stream Join

A *stream-stream* join returns a stream on its output. Additionally, we apply a sliding window on the input streams that effectively defines an event-time band-join over both:[6] two records join if they have the same join attribute value and if their timestamps are close to each other, i. e., their timestamp difference is less than or equal to the window size.

The *stream-stream join* operator `join`, takes as input two data streams S and $\bar{S}$, two extractor functions g and $\bar{g}$ that return the join attribute value for records of each stream, a window size parameter ω, as well as a joiner function j that computes the join result for two joining records. For each pair $r \in S$ and $\bar{r} \in \bar{S}$ that joins, j is called to compute the join result. A new record with the join result and timestamp $t' = \max\{r.t, \bar{r}.t\}$ is appended to the result stream.

Definition 67 (Windowed Stream-Stream Join)**.** *Given two record streams $S[\mathbb{S}]$ and $\bar{S}[\bar{\mathbb{S}}]$, two extractor functions $g : [\![\mathbb{S}]\!] \to \mathbb{D}_{K'}$ and $\bar{g} : [\![\bar{\mathbb{S}}]\!] \to \mathbb{D}_{K'}$, a window size parameter $\omega \in \mathcal{T}$, as well as a function $j : [\![\mathbb{S}]\!] \times [\![\bar{\mathbb{S}}]\!] \to \mathbb{D}_{V'}$, $\texttt{join} : S[\![\mathbb{S}]\!] \times S[\![\bar{\mathbb{S}}]\!] \times ([\![\mathbb{S}]\!] \to \mathbb{D}_{K'}) \times ([\![\bar{\mathbb{S}}]\!] \to \mathbb{D}_{K'}) \times \mathcal{T} \times ([\![\mathbb{S}]\!] \times [\![\bar{\mathbb{S}}]\!] \to \mathbb{D}_{V'}) \to S[\![\mathbb{S}']\!]$ with $\mathbb{S}' = \{O, T, \{O \times O\}, V'\}$ is defined as:*

$$\texttt{join}(S, \bar{S}, g, \bar{g}, \omega, j) = S' \tag{5.13}$$

with

$$\begin{aligned}
\forall r \in S, \forall \bar{r} \in \bar{S} : (\ & (g(r) = \bar{g}(\bar{r}) \wedge |r.t - \bar{r}.t| \leq \omega) \\
& \implies \exists r' \in S' : r' = \langle o, \max\{r.t, \bar{r}.t\}, \langle r.o, \bar{r}.o\rangle, j(r, \bar{r})\rangle\ ) \wedge \\
\forall r' \in S' : (\ & \exists r \in S, \exists \bar{r} \in \bar{S} : (r'.k = \langle r.o, \bar{r}.o\rangle \wedge g(r) = \bar{g}(\bar{r})\ \wedge |r.t - \bar{r}.t| \leq \omega) \wedge \\
& \forall r^* \in S' : (r^*.k = r'.k\ \implies\ r^* = r')\ )
\end{aligned}$$

Definition 67 explicitly assigns unique keys (as input record offset pair, instead of $\perp$) to the result records. The reason to assign keys explicitly is twofold: (1) Assigning keys explicitly is required for a correct definition of the result. Records at different offsets may have the same value and timestamp, leading to multiple result records with equal value and timestamp (at different offsets). If the definition would use $\perp$-keys, those "equal" result records could not be distinguished and the required number of join results would not be reflected in the definition because the $\exists$-quantifier cannot express quantities. Assigning unique keys based on input record offsets ensures that no join result is missing in the output stream. (2) Assigning keys explicitly allows us to handle unordered input data streams for outer joins

[6] Limiting the scope of a join is required to avoid unbounded growth of the operator state. Join operators usually buffer records in hash-tables to compute the join result incrementally [AA91, ACc+03b, Des04] (c. f. Section "Incremental Join Computation" below). Sliding-windows are the most common approach because it is easy to reason about the operator semantics. However, there also exist many other techniques [XY07].

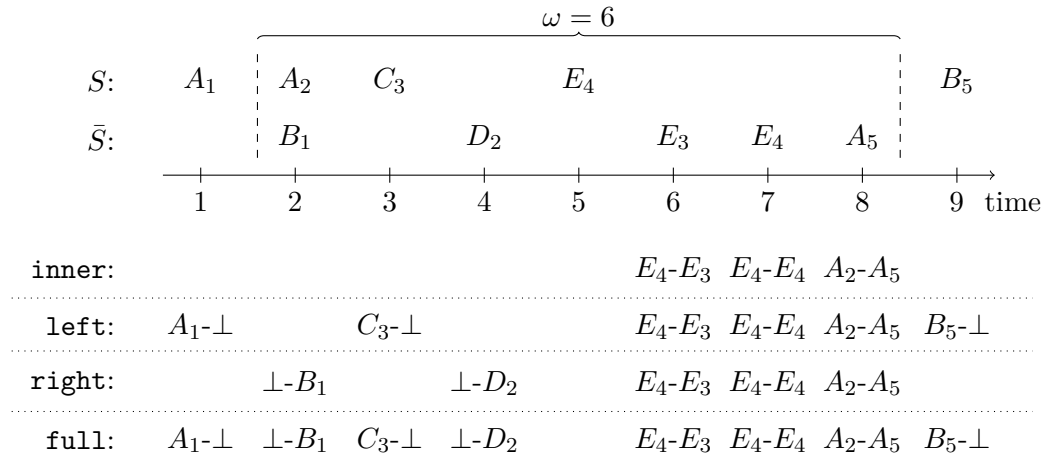

Figure 5.4: Stream-stream join example.

(as defined in the next paragraph) gracefully. We discuss joins on unordered input streams in more detail later (c. f. Section *"Joining Unordered Data Streams"*).

Furthermore, we point out that the offset attribute of result records is not explicitly defined, because any order of output records is considered correct. Similar to unary record stream transformations, we apply stream equivalence (Definition 60) as correctness criteria to stream-stream joins.

Outer-Joins: Additionally to the *inner* stream-stream join as introduced above (Definition 67), we also include *left-*, *right-* and *full-outer* joins in our model.[7] Their definition follows left-/right-/full-outer join semantics as known from relational database systems. If a record from the left and/or right input stream does not join on the *inner* join condition, i. e., no join partner record in the other streams exists, the record is still added to the result stream. We discuss later, how outer-joins are computed in more detail. For stream-stream joins, outer result records are computed as follows:

Left-Outer: $\langle o, r.t, \langle r.o, \bot\rangle, j(r, \bot)\rangle$

Right-Outer: $\langle o, \bar{r}.t, \langle \bot, \bar{r}.o\rangle, j(\bot, \bar{r})\rangle$

Example 19 (Stream-Stream Join)**.** *Figure 5.4 depicts two ordered input streams S and $\bar{S}$, and their inner/left-outer/right-outer/full-outer join result for a window of size $\omega = 6$, using the values as join attribute. We only show the values; their indices indicate the input record offsets. Timestamps are given by the timeline while keys, that are $\bot$, are omitted.*

The inner join produces 3 *join results. A_1 does not join with A_5 because they do not fall into the same join window. Similarly, the time difference between B_1 and B_5 exceeds the window size, and thus, both do not join. However, A_2 joins with A_5 and E_4 from S joins with E_3 and E_4 from $\bar{S}$. The other records (C_3 and D_2) within*

[7] We us the term *outer join* to refer the all three types of outer joins at once.

the time window do not join, because there is not record in the other stream with equal join attribute value.

The left-outer join contains all result records from the inner join. Additionally, the three left-stream records A_1, C_3, and B_5 are included in the result. Similarly, the right-outer join includes the two right-stream records B_1 and D_2 in addition to the inner join result. Finally, the full-outer join result contains all records from the inner-, left-outer-, and right-outer join results.

It is important to note that the depicted join result streams are shown in timestamp *order to indicate the assigned result record timestamps. The actual* offset *order may be different and is not specified in the example.*

To compute the correct result, a `join` operator needs to store records from both input streams in its internal state. If both input streams are ordered, the window size determines how long records needs to be stored. If the time in one input stream advances from t to $t+1$, all records from the other stream with timestamp up to $t-\omega$ can be deleted because they cannot join with any future records any longer. However, for the case of unordered input streams, it is a-priori unknown how long records need to be stored, because an out-of-order record might occur at any time. Conceptually, records would need to be stored forever, resulting in unbounded space requirements. Furthermore, for outer joins it is unclear if an outer join result can be emitted or not. Even if the join windows "ends", there might be an out-of-order record later that results in an inner join result. Because an out-of-order record may appear at any point in time, conceptually outer join results can never be added to the result stream safely, because it is unknown if they are contained in the result or not. We discuss both issues of unbounded space requirements and handling of outer join results for unordered input streams in more detail in Section 5.3.

Stream-Table Join

A *stream-table* join is a temporal table lookup-join and it yields an output stream as result. It is basically a "stream enrichment join": for each stream record, a join attribute is extracted and the join attribute is used to find a table record with the corresponding primary key. If such a record exists in the table, the stream record is joined with the table record and a join result is appended to the output stream. A stream-table join is a *temporal* join: the table lookup is based on the stream record join attribute *and* the stream record timestamp. In particular, a record is joined with the table version corresponding to the stream record timestamp.

The *stream-table join* operator `join`, takes as input a data stream S, a table $\vec{T}$, an extractor function g that returns the key from the stream record for the primary-key table lookup, as well as a joiner function j that computes the join result for two joining records. The return type of g must be the same as the primary key type of $\vec{T}$. Output records inherit the input stream record timestamp.

Definition 68 (Stream-Table Join)**.** *Given a record stream $S[\![\mathbb{S}]\!]$, a table $\vec{T}[\bar{\mathbb{T}}]$, an extractor function $g : [\![\mathbb{S}]\!] \to \mathbb{D}_{\bar{K}}$, and a function $j : [\![\mathbb{S}]\!] \times [\![\bar{\mathbb{T}}]\!] \to \mathbb{D}_{V'}$,* `join` *: $S[\![\mathbb{S}]\!] \times \vec{T}[\![\bar{\mathbb{T}}]\!] \times ([\![\mathbb{S}]\!] \to \mathbb{D}_{\bar{K}}) \times ([\![\mathbb{S}]\!] \times [\![\bar{\mathbb{T}}]\!] \to \mathbb{D}_{V'}) \to S[\![\mathbb{S}']\!]$ with $\mathbb{S}' = \{O, T, O, V'\}$ is defined as:*

$$\texttt{join}(S, \vec{T}, g, j) = S' \tag{5.14}$$

with

$$\begin{aligned}\forall r \in S : (\ & \bar{r} = T_{r.t}.\texttt{lookup}(r.k) \ \wedge \\ & (\bar{r} \neq \bot \implies \exists r' \in S' : r' = \langle o, r.t, r.o, j(r, \bar{r})\rangle)\ ) \ \wedge \\ \forall r' \in S' : (\ & \exists r \in S : (r.o = r'.k \wedge T_{r.t}.\texttt{lookup}(r.k) \neq \bot) \ \wedge \\ & \forall r^* \in S' : (r^*.k = r'.k \implies r^* = r')\ )\end{aligned}$$

It is important to note that Definition 68 explicitly assigns unique keys to the result records. The reason is the same as for stream-stream joins as discussed previously (Definition 67).

Example 20 (Stream-Table Join)**.** *Figure 5.5 depicts a stream-table join between a stream S and table $\vec{T}$. For S, only the record values that are used as join-attribute are shown. For $\vec{T}$, we also depict the modified key set in braces. The first stream record A at timestamp 1 does not join, because there is no corresponding table version. Inserting a key-value pair into the table at timestamp 2 does not yield a join result either, because there is no corresponding stream record. At timestamp 3, a new key-value pair $\langle B, 2\rangle$ is inserted into the table and a new stream record is processed. Hence, inner join result record B-2 is appended to the output stream. Stream record A with timestamp 4 joins with $\langle A, 1\rangle$ independent of the update of key B to value 3. At timestamp 5, stream record B is processed. Even if there is no table version at timestamp 5, B joins with the latest value and record B-3 is emitted, because a "missing" table version at timestamp 5 implies that $T_5 = T_4$. The last stream record A does not produce any join result because A is explicitly deleted from the table at the same time as indicated by the modified key set of T_5 that contains A.*

A stream-table join can also be a left-outer join ensuring that there is exactly one join result per stream input record.

Definition 69 (Left-Outer Stream-Table Join)**.** *Given a record stream $S[\mathbb{S}]$, a table $\vec{T}[\bar{\mathbb{T}}]$, an extractor function $g : [\![\mathbb{S}]\!] \to \mathbb{D}_{\bar{K}}$, and a function $j : [\![\mathbb{S}]\!] \times ([\![\bar{\mathbb{T}}]\!] \cup \{\bot\}) \to \mathbb{D}_{V'}$, $\texttt{leftJoin} : S[\![\mathbb{S}]\!] \times \vec{T}[\![\bar{\mathbb{T}}]\!] \times ([\![\mathbb{S}]\!] \to \mathbb{D}_{\bar{K}}) \times ([\![\mathbb{S}]\!] \times [\![\bar{\mathbb{T}}]\!] \to \mathbb{D}_{V'}) \to S[\![\mathbb{S}']\!]$ with $\mathbb{S}' = \{O, T, O, V'\}$ is defined as:*

$$\texttt{leftJoin}(S, \vec{T}, g, j) = S' \tag{5.15}$$

with

$$\begin{aligned}\forall r \in S : (\ & \bar{r} = T_{r.t}.\texttt{lookup}(r.k) \ \wedge \\ & \exists r' \in S' : r' = \langle o, r.t, r.o, j(r, \bar{r})\rangle\ ) \ \wedge \\ \forall r' \in S' : (\ & \exists r \in S : r.o = r'.k \ \wedge \\ & \forall r^* \in S' : (r^*.k = r'.k \implies r^* = r')\ )\end{aligned}$$

Example 20 includes the result of a left-outer stream-table join. For this case, every input stream record produces one output record. Right-outer and full-outer joins are not defined for stream-table joins, because the join is "asymmetric" and only stream-side records may result in output records, while table-side updates are applied "passively".

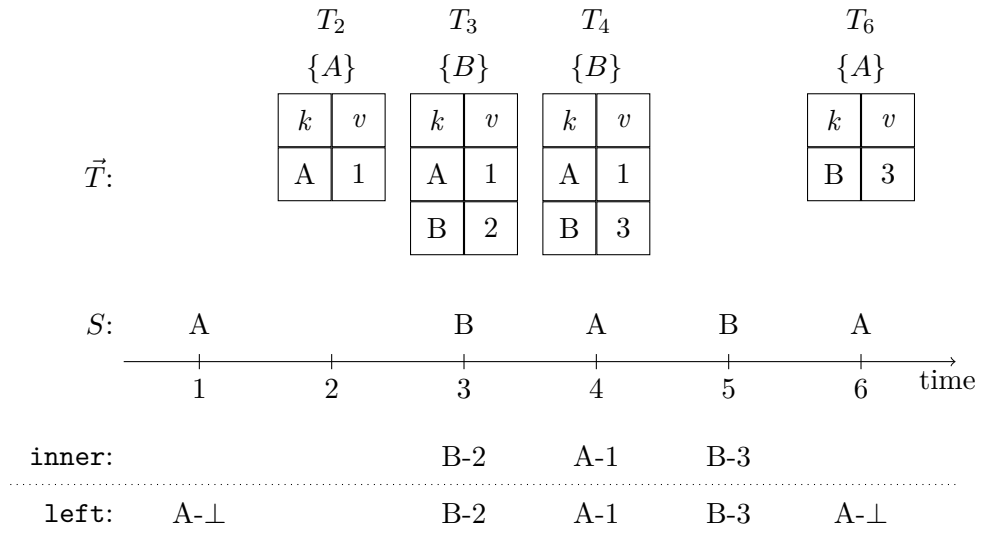

Figure 5.5: Stream-table join example.

Incremental Join Computation

Assuming ordered input data streams, stream-stream and stream-table joins can be computed incrementally. An incremental computation requires some time-synchronization between both inputs. Record processing alternates between the left and right input based on the record timestamps. For stream-table join we assume an ordered table-changelog stream as input that updates the right-hand side table accordingly. To synchronize both input streams, the operator always looks at the next record to be processed for both input streams, and picks the record with the smaller timestamp. If both records have the same timestamp, either record may be processed for a stream-stream join, while it is required to process a table changelog record before a stream record for stream-table joins (to ensure that the table is updated *before* a table-lookup is performed).

For example, the inner stream-stream join can be implemented as symmetric-hash-join [AA91, ACc+03b, Des04]: a left record is processed by inserting it into the left hash-table and additional probing of the right hash-table. Since joins are monotone operators [TGNO92], processing records incrementally in offset-order yields the correct result.

Computing inner stream-stream and inner/left-outer stream-table joins incrementally, using input stream time-synchronization as described above, produces ordered output data streams. For stream-stream outer joins it is required to apply a more sophisticated algorithm, because emitting outer join results must be delayed until window size time passes (otherwise, a record might still produce an inner join result). If it is required that the output stream is ordered, delaying outer join result records requires delaying inner join result records, too. In such case, all computed result records would be buffered in the operator state first. While time advances, inner and outer join result records with a timestamp smaller than the current time minus the window size can be appended to the output stream in timestamp order.[8]

[8]Instead of buffering the result records, it is also possible to delay the whole computation until window size time has passed. For each buffered record that is ready for processing, i. e., its time-

Joining Unordered Data Streams

Handling out-of-order records in stream-stream and stream-table join inputs requires several strategies. For inner stream-stream joins, it is sufficient to preserve history, while for outer stream-stream and stream-table joins preserving history is necessary but not sufficient to compute the correct result. The decision if a result record needs to be appended to the output stream is not affected for inner stream-stream joins if inputs are unordered. If two records join, it is always correct to emit a result record. As long as stream history is preserved, i. e., older records are buffered in the operator state, any out-of-order record can be joined correctly.[9] However, for outer stream-stream joins and left-outer stream-table join, if a record does not have any join partner in the other input, it is unclear if emitting an outer join result is correct, because a future out-of-order record in the other input might result in an inner join result later.

Consider Figure 5.4 from Example 19: the first record A_1 from left input stream S produces a left-outer join result record A_1-$\perp$. Assuming an incremental join computation (c. f. Section "Incremental Join Computation" from above), it is unknown when processing A_1 if there might be a A-record with timestamp 2 to 7 (or actually with timestamp -5 to 7, as earlier and later records need to be considered) in the right input stream after record B_1. The naïve strategy to delay emitting A_1-$\perp$ after a record with timestamp 8 is observed on $\bar{S}$ only works for ordered input stream. If there is no out-of-order data, it is clear that there would not be any joining right hand side record and A_1-$\perp$ is the correct result. However, for unordered data streams, there might be a joining record at any point in the future and emitting A_1-$\perp$ might be incorrect. Some systems apply punctuation/watermarks [TMSF03, ABB$^+$13] to address this problem, however, we argue (c. f. Section 5.3) that this couples processing latency to data properties, which is undesirable. Processing latency should depend on the system and its processing/operator model, but not on the processed data. Therefore, we suggest an *emit-eager-and-update-later* processing strategy. Each time a record is processed and no inner join result is computed, the operator emits an outer join result immediately. If a inner join result is computed later, it is treated as an *update* to the previous outer join result. Hence, instead of emitting a second *fact* record, update semantics similar to table changelog streams are applied.

Example 21 (Eager Join Compuation)**.** *Figure 5.6 depicts the same input streams as Figure 5.4 from Example 19, and the result of a left- and right-outer join with the same window size $\omega = 6$. Additionally, the output record keys are shown. The result is computed eagerly, and hence, each time a record is processed an output record is emitted.*

For the left-outer join, all records from the left input stream produce a left-outer join result, because no matching right hand side record was processed yet. Processing records B_1 and D_2 from $\bar{S}$ does not result in any output records, as no matching left hand side record is available, and for a left join right input records do not trigger

stamp is smaller than current time minus window size, the probing into the other hash-table can be done considering only *later* records with respect to the currently processed record. *Earlier* records are not considered, because the corresponding join results would have been computed already when those earlier records became available for processing.

[9] It is important to note that it is impossible to guarantee ordered output streams if input streams are unordered.

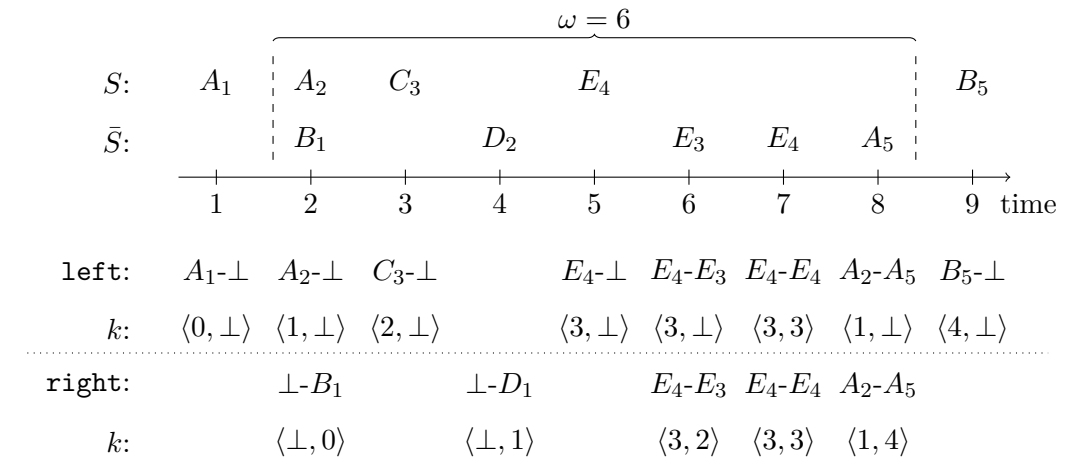

Figure 5.6: Stream-stream left- and right-outer join example with eager emitting.

any outer join results. However, processing E_3, E_4, and A_2 from $\bar{S}$ produces inner join result records. The first result E_4-E_3 is an update to the previously emitted left join results E_4-$\perp$ and hence gets the same key $\langle 3, \perp \rangle$ assigned. The second result E_4-E_4 does not need to update any left join result and hence it gets it own unique key based on the input record offsets $\langle 3, 3 \rangle$ assigned. Similar to the first record, the last record A_5 gets the same key $\langle 1, \perp \rangle$ assigned as the left join result from timestamp 2 to "update" the join result from an outer join result to an inner join result.

For the right-outer join, the exact same output as in Example 19 is emitted. In our example, no "incorrect" eager right-outer join result is emitted and hence there is nothing to be updated.

Example 21 illustrates our emit-eager-and-update-later processing strategy. The emitted output streams, are a super set of the expected output streams as illustrated in Figure 5.4. It is important to note that the result is no longer a record stream (Definition 53), because it contains *updates*. Hence, record stream equivalence (Definition 60) does not apply. We redefine equivalence for data streams with updates, which allows us to apply our operator correctness definition (Definition 61).

Definition 70 (Equivalence of Data Streams With Updates)**.** *Given a stream $S[\mathbb{S}]$ that contains updates. $S[\mathbb{S}]$ can be transformed into a* record *stream $S'[\mathbb{S}]$, by removing all records that are updated later:*

$$S' = S.\texttt{filter}(f)$$
$$\textit{with:} \quad f(r) = \begin{cases} \top & \textit{if } \nexists \bar{r} \in S : (\bar{r}.k = r.k \wedge \bar{r}.o > r.o) \\ \bot & \textit{otherwise} \end{cases}$$

Given two streams $S_1[\mathbb{S}]$ and $S_2[\mathbb{S}]$ that both may contain updates. Both streams are equivalent, denoted $S_1[\mathbb{S}] \equiv S_2[\mathbb{S}]$, iff their corresponding record streams are equivalent:

$$S_1 \equiv S_2 \iff \texttt{filter}(S_1, f) \equiv \texttt{filter}(S_1, f) \tag{5.16}$$

Definition 70 states that two streams containing updates are equivalent if they contain the same data after all "outdated" records are removed from the stream.

Considering Example 19 and Example 21, both left-outer join result streams are equivalent because the additional records A_2-$\perp$ and E_4-$\perp$ from the eager join computation are updated via A_2-A_5 and E_4-E_3 and hence the record stream that corresponds to the eager result stream (Definition 70) is equivalent with the result stream from Example 19.

Applying our eager-computation strategy, handling out-of-order records for outer stream-stream joins is achieved natively.

Example 22 (Joining Unordered Streams)**.** *Figure 5.7 uses equivalent input streams as Example 19 and Example 21, however, records are out-of-order (highlighted in red). Therefore, no timeline is depicted but records are ordered by their offsets and timestamps are added explicitly to each record (i. e., $\langle t, v \rangle$)*[10]*. Both input streams are shown interleaved to indicate the processing order, which is based on the record timestamps as discussed above (c. f. Section "Incremental Join Computation"). It is important to note that the lower part of the figure is a continuation of the upper part. As before, the join window has a size of $\omega = 6$ (not depicted).*

The inner join result is the same as in the previous examples illustrating that inner stream-stream joins are not affected by out-of-order data and produce result record streams (i. e., no updates are required). However, the result stream is unordered as compared to previous examples because the input streams are unordered. For the left-outer join, there is an eager result from processing A_2 at offset 2 in S that is updated later with A_2-A_5 when A_5 from $\bar{S}$ is processed (highlighted in blue). This is similar to Example 21. In contrast, because E_4 from S is out-of-order and is processed after E_3 and E_4 from $\bar{S}$, no eager outer join result is contained in the left-outer result stream for E-records, but only the two inner join results. At the same time, there are two outer join results for E_3 and E_4 in the right-outer join result stream (highlighted in green), due to the delay of E_4 in S. Both are updated when E_4 is processed as last record of S resulting in two output records, one for each previous outer join result. The full-outer join result contains all eager left- and right-outer join result records as expected. It is important to note that the keys of the records are assigned differently in each case, depending if an inner join result is an update to a previous eager outer join result or not. Finally, the fact that record D_2 from $\bar{S}$ is out-of-order does not have an impact on the result: because it results in a correct right-outer join result, it does not matter when D_2 is processed.

Example 22 illustrates that handling out-of-order records is natively achieved applying our eager-emit-and-update strategy. The order of input records as well as their delay only affects which intermediate outer join result records are computed, but do not have an impact on the final result.

For stream-table joins, out-of-order stream records do not require special handling. For each record, a table lookup is done and a corresponding output record is appended to the output stream if appropriate. However, out-of-order table updates could yield incorrect join results, if not treated properly. Assume that the table update in Figure 5.5 from $\langle B, 2 \rangle$ to $\langle B, 3 \rangle$ is delayed. Stream record B at timestamp 5 would incorrectly join with the table version 3 and emit $\langle B, 2 \rangle$. To handle this case,

[10] The value indices are the same as in the previous examples to allow an easy mapping between them.

offset:	0		1	2		
S:	$\langle 1, A_1 \rangle$		$\langle 3, C_3 \rangle$	$\langle 2, A_2 \rangle$		
$\bar{S}$:		$\langle 2, B_1 \rangle$			$\langle 6, E_3 \rangle$	$\langle 4, D_2 \rangle$
offset:		0			1	2
inner:						
left:	$\langle 1, \langle 0, \bot \rangle, A_1\text{-}\bot \rangle$		$\langle 3, \langle 1, \bot \rangle, C_3\text{-}\bot \rangle$	$\langle 2, \langle 2, \bot \rangle, A_2\text{-}\bot \rangle$		
right:		$\langle 2, \langle \bot, 0 \rangle, \bot\text{-}B_1 \rangle$			$\langle 6, \langle \bot, 1 \rangle, \bot\text{-}E_3 \rangle$	$\langle 4, \langle \bot, 2 \rangle, \bot\text{-}D_2 \rangle$
full:	$\langle 1, \langle 0, \bot \rangle, A_1\text{-}\bot \rangle$	$\langle 2, \langle \bot, 0 \rangle, \bot\text{-}B_1 \rangle$	$\langle 3, \langle 1, \bot \rangle, C_3\text{-}\bot \rangle$	$\langle 2, \langle 2, \bot \rangle, A_2\text{-}\bot \rangle$	$\langle 6, \langle \bot, 1 \rangle, \bot\text{-}E_3 \rangle$	$\langle 4, \langle \bot, 2 \rangle, \bot\text{-}D_2 \rangle$

offset:			3	4		
S:			$\langle 9, B_5 \rangle$	$\langle 5, E_4 \rangle$		
$\bar{S}$:	$\langle 8, A_5 \rangle$	$\langle 7, E_4 \rangle$				
offset:	3	4				
inner:	$\langle 8, \langle 2, 3 \rangle, A_2\text{-}A_5 \rangle$			$\langle 6, \langle 4, 1 \rangle, E_4\text{-}E_3 \rangle$	$\langle 7, \langle 4, 4 \rangle, E_4\text{-}E_4 \rangle$	
left:	$\langle 8, \langle 2, \bot \rangle, A_2\text{-}A_5 \rangle$		$\langle 9, \langle 3, \bot \rangle, B_5\text{-}\bot \rangle$	$\langle 6, \langle 4, 1 \rangle, E_4\text{-}E_3 \rangle$	$\langle 7, \langle 4, 4 \rangle, E_4\text{-}E_4 \rangle$	
right:	$\langle 8, \langle 2, 3 \rangle, A_2\text{-}A_5 \rangle$	$\langle 7, \langle \bot, 4 \rangle, \bot\text{-}E_4 \rangle$		$\langle 6, \langle \bot, 1 \rangle, E_4\text{-}E_3 \rangle$	$\langle 7, \langle \bot, 4 \rangle, E_4\text{-}E_4 \rangle$	
full:	$\langle 8, \langle 2, \bot \rangle, A_2\text{-}A_5 \rangle$	$\langle 7, \langle \bot, 4 \rangle, \bot\text{-}E_4 \rangle$	$\langle 9, \langle 3, \bot \rangle, B_5\text{-}\bot \rangle$	$\langle 6, \langle \bot, 1 \rangle, E_4\text{-}E_3 \rangle$	$\langle 7, \langle \bot, 4 \rangle, E_4\text{-}E_4 \rangle$	

Figure 5.7: Stream-stream join example for unordered input streams with $\omega = 6$.

we apply the same emit-eager-and-update-later incremental processing strategy. To allow updating result records later, it is required to buffer input stream records in the stream-table join operator and re-trigger the join computation for out-of-order table updates. If an out-of-order table update occurs, corresponding update records are sent downstream to update previously emitted join records.

5.2.4 Table Operators

We define table operators with relational semantics enriched with a temporal component to incorporate the nature of *evolving* tables (Definition 56) that have multiple table versions. Additionally, we limit the scope of allowed transformations to `mapValues` (i. e., key-preserving projection as introduced in Definition 71 below), `filter` (i. e., selection), aggregation, and equi-join. Those operators allow us to maintain result tables incrementally if the input tables are updated, using techniques from relational database systems to maintain materialized views.

Since input tables are updated continuously, a table operator needs to continuously update the result table. Therefore, result tables are effectively materialized views. To apply updates with low latency as required for a stream processing model, materialized views need to be updated *incrementally* [BLT86, JMS95, LPBZ96]. Hence, all known limitations to allowed table operations apply in our model. For example, aggregation functions need to be subtractable to avoid unbounded space requirements for an incremental computation.

In practice, not all tables need to be materialized and some computations may be performed on a table changelog stream (for example a filter). A cost model shall be employed to decide if an operator uses a materialized table or operates over the corresponding table changelog stream only.

Table Transformations

The *mapValues* operator `mapValues` is a second-order function that takes a table and a user-defined first-order function $f : [\![\mathbb{T}]\!] \to \mathbb{D}_{V'}$ as parameters. For each record in table $\vec{T}$, `mapValues` invokes f and inserts a record with returned value into the output table.

Definition 71 (MapValues Operator)**.** *Given a table* $\vec{T}[\mathbb{T}]$ *and a function* $f : [\![\mathbb{T}]\!] \to \mathbb{D}_{V'}$*. We define* $\texttt{mapValues}(\vec{T}, f) : \vec{T}[\![\mathbb{T}]\!] \times ([\![\mathbb{T}]\!] \to \mathbb{D}_{V'}) \to \vec{T}[\![\mathbb{T}']\!]$ *with* $\mathbb{T}' = \{T, K, V'\}$ *as follows.*

$$\texttt{mapValues}(\vec{T}, f) = (T'_0, \ldots, T'_{\hat{t}}) \tag{5.17}$$

with

$$\vec{T} = (T_0, \ldots, T_{\hat{t}}) \wedge \forall T_t \in \vec{T} : T'_t = \texttt{mapValues}(T_t, f)$$

and

$$\texttt{mapValues} : T[\![\mathbb{T}]\!] \times ([\![\mathbb{T}]\!] \to \mathbb{D}_{V'}) \to T[\![\mathbb{T}']\!]$$
$$\texttt{mapValues}(T, f) = T'$$

with

$$\begin{aligned} &\forall r \in T : (\exists r' \in T' : r' = \langle r.t, r.k, f(r)\rangle) \wedge \\ &\forall r' \in T' : (\exists r \in T : r'.k = r.k) \end{aligned} \tag{5.18}$$

The `mapValues` operator preserves key and timestamp for each record. We do not allow to modify the key, because this may lead to primary key conflicts on the resulting table that cannot be resolved deterministically. Assume an input table with two records $\langle k_1, t, v_1 \rangle$ and $\langle k_2, t, v_2 \rangle$, and a function $f(r) = \langle k, r.v \rangle$. Because there is no order between both input table records, it is not defined if the result table should contain $\langle k, t, v_1 \rangle$ or $\langle k, t, v_2 \rangle$. If a table shall be re-keyed, it is required to use the aggregation operator (c. f. Definition 73 below).

To express a filter/selection on tables, we use the `mapValues` operator and allow f to return $\perp$ (or return the value unmodified) to indicate that a record is dropped. To incorporate $\perp$ as return value, Definition 71 is updated to:

$$\texttt{mapValues}(T, f) = T'$$

with

$$\forall r \in T : v' = f(r) \wedge (v' \neq \perp \implies (\exists r' \in T' : r' = \langle r.t, r.k, v' \rangle)) \wedge$$
$$\forall r' \in T' : (\exists r \in T : r.k = r'.k \wedge f(r) \neq \perp)$$

To allow users to only specify a boolean predicate that returns `true` or `false` instead of a function f that can be provided to `mapValues`, we define the *filter* operator `filter` as second-order function that takes a user-defined filter predicate as input as follows.

Definition 72 (Filter Operator)**.** *A* filter *operator is a second-order function* $\texttt{filter} : \vec{T}[\![\mathbb{T}]\!] \times ([\![\mathbb{T}]\!] \to \{\perp, \top\}) \to \vec{T}[\![\mathbb{T}]\!]$*. It takes a table* $\vec{T}$ *and a user-defined filter predicate* $p : [\![\mathbb{T}]\!] \to \{\perp, \top\}$*, and applies* p *to each record in the table. The result table contains all records for which* p *returns* $\top$*.*

$$\texttt{filter}(\vec{T}, p) = \texttt{mapValues}(\vec{T}, \textit{mapValuesFilter}(p)) \tag{5.19}$$

with

$$\textit{mapValuesFilter} : ([\![\mathbb{T}]\!] \to \{\perp, \top\}) \to ([\![\mathbb{T}]\!] \to ([\![\mathbb{T}]\!] \cup \{\perp\}))$$

$$\textit{mapValuesFilter}(p) = r \to f(r)$$

$$\textit{with } f(r) = \begin{cases} \perp & \textit{if } p(r) = \perp \\ r.v & \textit{if } p(r) = \top \end{cases}$$

The helper function `mapValuesFilter` in Definition 72 takes a filter predicate and returns a function that can be provided to `mapValues`.

Table Aggregation

The *aggregation* operator `agg` is a second-order function that takes a table, a user-defined first-order grouping function $g : [\![\mathbb{T}]\!] \to \mathbb{D}_{K'}$, and a user-defined first-order aggregation function $f : T[\![\mathbb{T}]\!] \to \mathbb{D}_{V'}$ as input, and produces a table as output. The input table is split into subsets based on g, i. e., one subset per grouping attribute value. The aggregation function f is applied to each subset to compute the aggregation result.

Definition 73 (Aggregation Operator)**.** *Given a table* $\vec{T}[\mathbb{T}]$*, a function* $g : [\![\mathbb{T}]\!] \to \mathbb{D}_{K'}$*, and a function* $f : T[\![\mathbb{T}]\!] \to \mathbb{D}_{V'}$*. The* aggregation *operator* $\texttt{agg} : \vec{T}[\![\mathbb{T}]\!] \times ([\![\mathbb{T}]\!] \to \mathbb{D}_{K'}) \times (T[\![\mathbb{T}]\!] \to \mathbb{D}_{V'}) \to \vec{T}[\![\mathbb{T}']\!]$ *computes a table* $\vec{T}$ *with schema* $\mathbb{T}' = \{T, K', V'\}$ *as:*

$$\texttt{agg}(\vec{T}, g, f) = (T'_0, \cdots, T'_{\hat{t}}) \tag{5.20}$$

with

$$\vec{T} = (T_0, \ldots, T_{\hat{t}}) \wedge \forall T_t \in \vec{T} : T'_t = \texttt{agg}(T_t, f, g)$$

Definition 73 uses the relational aggregation operator and applies it to all table versions to incorporate the temporal nature of an evolving table.

Table-Table Joins

Joining two tables requires an equi-join condition (as all other joins), and computes an output table. Similar to stream-table joins, table-table joins are *temporal* joins, implying that table versions with the same version number are joined.

The *join* operator `join`, takes as input two tables, two extractor functions g_1 and g_2 that return the join key value from table records for each input table, as well as a joiner function j that computes the join result for two joining records.

Definition 74 (Table-Table Join)**.** *Given a table* $\vec{T}_1[\mathbb{T}_1]$*, a table* $\vec{T}_2[\mathbb{T}_2]$*, two functions* $g_1 : [\![\mathbb{T}_1]\!] \to \mathbb{D}_k$ *and* $g_2 : [\![\mathbb{T}_2]\!] \to \mathbb{D}_k$*, as well as a function* $j : [\![\mathbb{T}_1]\!] \times [\![\mathbb{T}_2]\!] \to \mathbb{D}_{V'}$*,* $\texttt{join} : \vec{T}[\![\mathbb{T}_1]\!] \times \vec{T}[\![\mathbb{T}_2]\!] \times ([\![\mathbb{T}_1]\!] \to \mathbb{D}_k) \times ([\![\mathbb{T}_2]\!] \to \mathbb{D}_k) \times ([\![\mathbb{T}_1]\!] \times [\![\mathbb{T}_2]\!] \to \mathbb{D}_{V'}) \to \vec{T}[\![\mathbb{T}']\!]$ *with* $\mathbb{T}' = \{T, \langle K_1, K_2\rangle, V'\}$ *joins both tables by joining their corresponding table versions:*

$$\texttt{join}(\vec{T}_1, \vec{T}_2, g_1, g_2, j) = \vec{T}' = (T'_0, \cdots, T'_{\hat{t}}) \tag{5.21}$$

with

$$\hat{t} = \max\{t | T_t \in \vec{T}_1 \vee \bar{T}_t \in \vec{T}_2\} \wedge$$
$$\forall T'_t \in \vec{T}' : \forall r \in T_t, \forall \bar{r} \in \bar{T}_t :$$
$$g_1(r) = g_2(\bar{r}) \implies \langle \max\{r.t, \bar{r}.t\}, \langle r.k, \bar{r}.k\rangle, j(r, \bar{r})\rangle \in T'_t$$

Definition 74 defines an inner equi-join on the join attribute value returned by g_1 and g_2, and hence is a many-to-many join. For one-to-many or one-to-one joins, the definition could be adapted accordingly. Furthermore, defining left-, right-, and full-outer joins is straightforward. Even if we omit the corresponding formal definitions, our model supports all those joins. All joins follow the same temporal semantics that we illustrate for inner equi-joins in the following example.

Example 23 (Table-Table Join)**.** *Figure 5.8 depicts an inner join example for two input tables* $\vec{T}_A$ *(with table versions* 1*,* 5*,* 6*) and* $\vec{T}_B$ *(with table versions* 2*,* 3*,* 6*). Similar to a stream-table join if a table version is missing, it implies that there was no update with corresponding timestamp. Hence, we can use the previous table version because* $T_t = T_{t-1}$ *for this case.*

In our example, there is no result table version T'_1 *because there is no version* $T_1^{(B)}$ *and thus* $T_1^{(A)}$ *cannot be joined at timestamp* 1*. Because* $T_2^{(A)}$ *does not exist,* $T_2^{(B)}$ *joins with* $T_1^{(A)}$ *yielding* T'_2*. Updating* $T_2^{(B)}$ *to* $T_3^{(B)}$ *results in updating* T'_2 *to* T'_3*. Similarly,* T'_5 *is the result of joining the updated* $T_5^{(A)}$ *with* $T_3^{(B)}$*. At timestamp* 6 *both input tables are updated at the same time and thus* $T'_6 = T_6^{(A)} \bowtie T_6^{(B)}$*.*

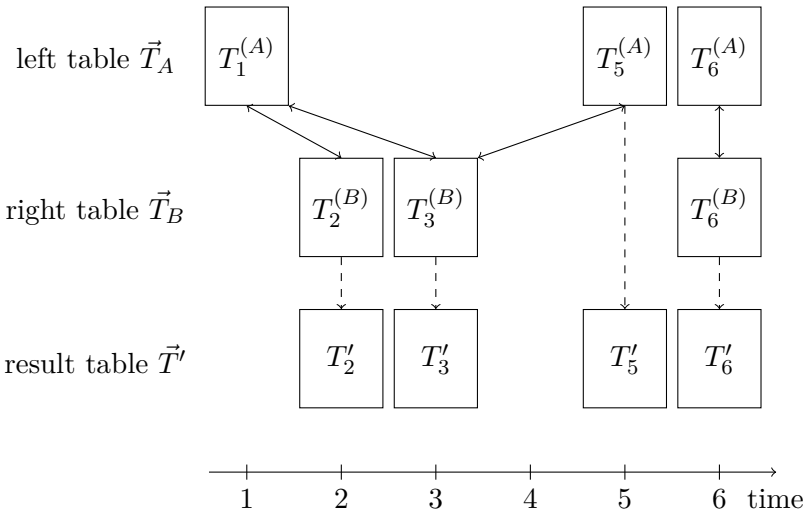

Figure 5.8: Table-table join example.

5.3 Model Trade-offs

Our *Dual Streaming Model* embraces the notion of "the result so far" [LWZ04], incremental record-by-record processing, and handles out-of-order records leveraging "updates". The main goals of our model are, to decouple the processing of out-of-order records from the processing latency, and to open up the design space for stream processing applications. To this extent, our model does not require external time tracking or time estimation mechanism like punctuation [TMSF03] or watermarks [ABB+13], but addresses out-of-order records *within* the processing model itself. However, punctuation/watermark may still be used to achieve different trade-offs within the model, as we discuss in more detail later.

Figure 5.9 depicts the 3-dimensional design space for stream processing models incorporating result correctness/completeness, processing cost, and processing latency. Result correctness/completeness targets the processing of unordered data streams. Depending on the maximum unorder/delay of out-of-order input records, it is required to buffer a certain history of data (either raw input records or partial results) to compute the correct result, which increases the processing cost. While the design space is 3-dimensional, existing stream processing models often limit users to have only a trade-off between processing cost vs. result correctness/completeness. Those models only support immutable results, and hence, couple processing latency to result correctness/completeness. Results can only be emitted after it is *known*[11] that all corresponding input data was received.

For example, buffering techniques that order data streams before processing (green line in Figure 5.9) as well as punctuations/watermarks (orange line in Figure 5.9) imply an increased processing latency if result correctness/completeness shall be achieved. The higher the maximum unorder/delay of out-of-order input records is, the longer data needs to be buffered and the later a punctuation/watermark arrives. Hence, processing cost and processing latency increase at the

[11] If all data was received or not, is often an estimation.

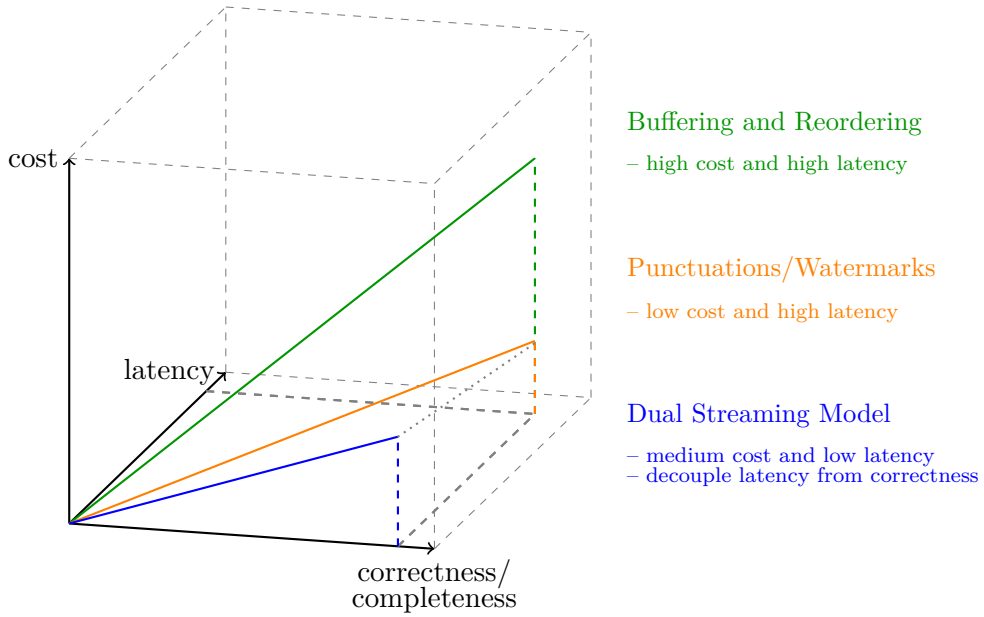

Figure 5.9: Trade-offs of data stream processing models.

same time. To this end, record buffering and reordering implies higher costs compared to a punctuation/watermark approach, as noted by Li et al. [LTS+08].

In contrast, our model (blue line in Figure 5.9) always achieves low processing latency, even if processing costs increase to achieve higher result correctness/completeness. Independent of the maximum unorder/delay of records, data is processed in offset order and the result is updated immediately by either updating a result table or appending a result stream record eagerly. We discuss latency in watermark based models compared to our model in more detail in the next section using the example of the window aggregation operator.

5.3.1 Processing Latency

Most stream processing models are "stream-only" models implying that operator inputs and outputs are always immutable record streams. In contrast to our model, an aggregation does not yield a result table, but a result stream in those models:

$$\text{record-stream} \rightarrow \text{aggregation-operator} \rightarrow \text{record-stream}$$

For a windowed aggregation, those models imply that there is exactly one output record per window per grouping attribute-value in the result stream. To compute the aggregation result, the aggregation operator maintains internal state, most likely as a key-value store. For example, for each window and each grouping attribute-value, a unique key/window-ID is generated (c. f. Section "Windowed Aggregation" above). For each window-ID, a corresponding entry is added to the key-value store. The value can either be a list of all raw input records that belong to the window that is identified by the window-ID, or an already pre-aggregated representation of the aggregation result. When a corresponding watermark arrives, the system triggers the computation of the aggregation and appends a result record to the output stream.

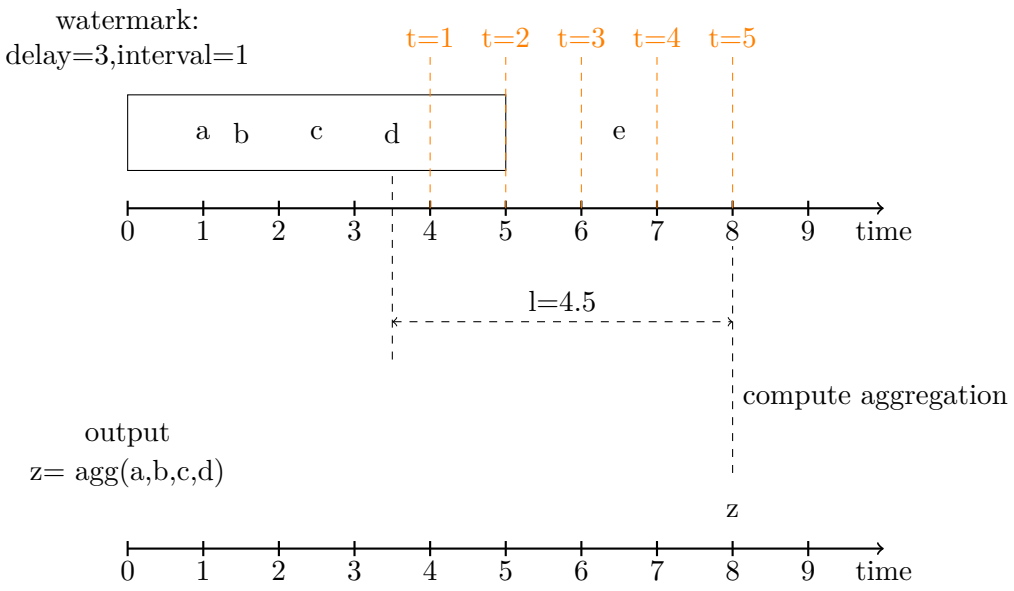

Figure 5.10: Windowed aggregations with watermarks.

Additionally, the corresponding key-value entry is removed from the internal state, because the computation for the window is completed and the window is not needed any longer.

Figure 5.10 depicts the computation of an aggregation window with window size $\omega = 5$, a watermark delay of 3, and watermark generation interval of 1 time unit. The first window has boundaries $[0, 5)$ and hence it is closed when a watermark with timestamp $t = 5$ is observed. Such a watermark arrives at timestamp 8 triggering the computation of the aggregation result z that is appended to the output stream. The processing latency is $l = 4.5$ time units, because the result could have been computed at time 3.5 when the last record d in the window arrived.

In our model, the result of an aggregation is not a record stream but a table plus its changelog stream:

$$\text{record-stream} \rightarrow \text{aggregation-operator} \rightarrow \text{table plus changelog}$$

For each input record, the aggregation operator updates the result table and appends a corresponding changelog record to the output stream. Figure 5.11 depicts the same windowed aggregation as Figure 5.10, using our processing model, showing the result table changelog stream. Each time a new input record is received, the aggregation result is updated (with zero latency $l = 0$) and a changelog record is emitted. Hence, the final window result is emitted at timestamp 3.5, the same time the last record d within the window arrives. In our model, the result latency (ignoring the actual computation time) is zero. The main difference of both results is that the changelog stream contains 4 output records that "update" each other, while the record stream contains one result record. Because the changelog stream contains more output records, the processing cost for downstream operators are higher compared to the watermark model. This property is reflected in Figure 5.9,

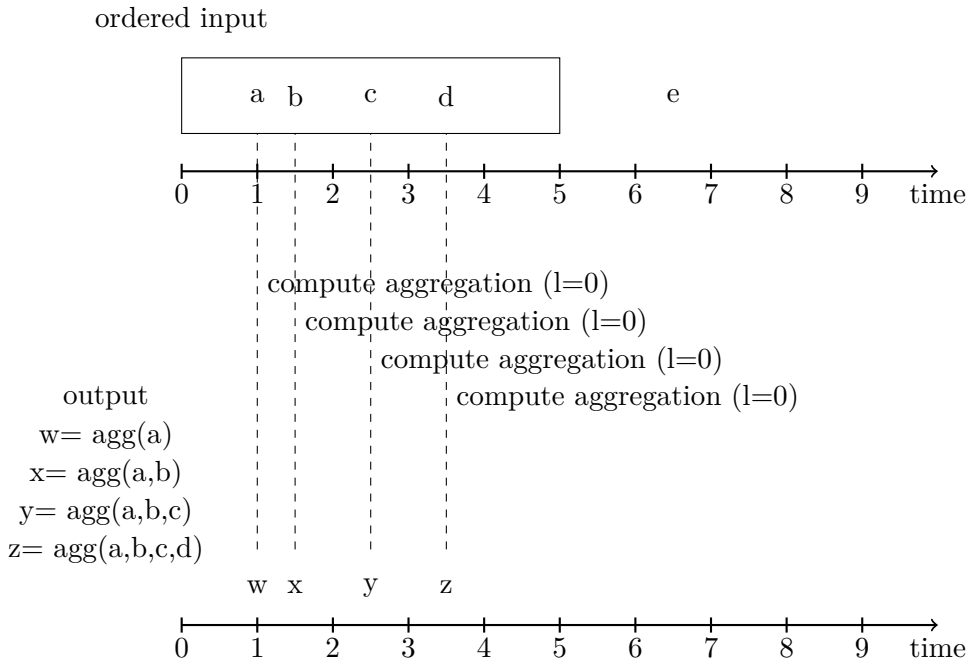

Figure 5.11: Continuous windowed aggregation.

and our model is considered to provided medium processing costs, compared to low processing cost as the watermark model. However, result latency is decoupled from record correctness/completeness.

Considering unordered input data streams, the processing latency in the watermark model does not depend on the event-time of the last record in a window, instead it depends on the difference between the processing timestamps of the last out-of-order record and the processing timestamp of the watermark that triggers the computation. How large the latency may be is data and hence application dependent. It is important to note that the latency may vary significantly between result records. In the watermark model, the computation of all parallel windows (i. e., same time boundaries but different aggregation attribute-values) is triggered by a single watermark. Hence, the latency is different for each parallel window. Furthermore, the unorder/delay distribution of records impacts the processing latency in the watermark model. If the distribution is skewed, i. e., there is a very small fraction of records that have very high unorder/delay, it is required to set the watermark high as well increasing the latency for most records significantly.

Figure 5.12 depicts an aggregation with 6 parallel windows that end at timestamp 5 (windows not shown), aggregation attributes a to f, and a corresponding out-of-order record per window. The timeline depicts processing-time, and we assume that all records have an event-timestamp between $[0, 5)$ and belong to the corresponding window. The watermark delay is set to 10 and hence the watermark with timestamp 5 that closes the windows arrives at timestamp 15. Only record e arrives with a high unorder/delay and is close to the watermark. For this record, the latency is low (note that the latency does not depend on the window end time). All other

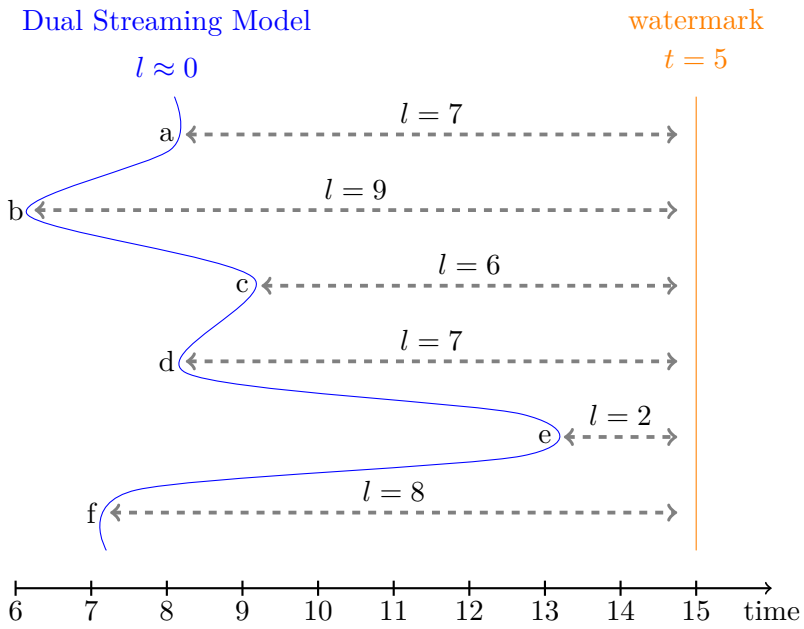

Figure 5.12: Processing latency in the watermark model.

records are much earlier and their latency is high, because the unorder/delay is not evenly distributed over all records. If the unorder/delay distribution is skewed, watermarks provide a poor upper bound on time progress. In contrast, our model does not depend on watermarks and thus latency for each window is independent as indicated by the blue line in Figure 5.12. Hence, the Dual Streaming Model allows to compute the final result as soon as the last record is available, while a watermark based approach needs to wait for a corresponding watermark to arrive. Hence, even if the maximum unorder/delay increases, processing latency stays low in our model, while it increases in the watermark model, based on the record unorder/delay distribution.

5.3.2 Design Space

Our model decouples processing latency from result correctness, however, it increases the load of downstream operators for certain type of computations. For example, the aggregation operator emits one output record per input record for non-windowed and tumbling-window aggregations. For overlapping windows, each input record updates multiple windows and hence results in multiple output records. A watermark based model emits only one result record per window per aggregation attribute-value, and hence, reduces the downstream load significantly. In a distributed environment, a watermark based model may also use upstream pre-aggregation before data is redistributed based on the aggregation attribute. Pre-aggregation cannot be applied in our continuous model though, because our model would required to send the current partial pre-aggregation result downstream immediately to allow an immediate update of the aggregation result. Therefore, no load reduction can be achieved using upstream pre-aggregation.

The above paragraph describes our model in its *pure* form that is latency optimized. It is easily possible to enhance our model with different emit strategies. For example, it would be possible to emit the first result of a windowed aggregation only after the window end time passed, and update the window immediately for every consecutive update. This strategy reduces the downstream load significantly if an application is not interested in early (and partial) window aggregation results. For this case, upstream pre-aggregation may be applied as long as window end time did not pass. Some watermark based models also allow for flexible triggering strategies [ABC+15]. However, they do not define update semantics in result streams but treat outputs as record streams. Hence, triggering a window computation multiple times may lead to incorrect downstream results.

Similar to windowed aggregations, it is possible to delay the computation of outer-join result records until the window end time passed. If each record is processed eagerly, the probability of "false" outer-join results that are updated later is high, because not all records that belong to the join window are received yet. The drawback of delaying the computation of outer-join result is an increased processing latency for all actual outer-join result records. It is important to note that the latency of inner-join result records is not affected. Hence, users may chose between an eager and lazy emit strategy depending on the number of expected outer-join result records.

Additionally, our model allows applications to retrieve result tables either push or pull based. Because tables are a first class citizen, applications can exploit the duality between changelog streams and tables, and either subscribe to a table changelog stream (push) or query the result via table lookups (pull). Furthermore, it is possible that different applications request different emit strategies. For example, two application might be interested in a certain data stream aggregation result. The first application contains a dash-board and is interested in every update to display the partial results. At the same time, the second application may only be interested in final results. For this case, the computation can be done once, and two different output streams are published using different emit strategies. In contrast, watermark based models couple the emit strategy to the aggregation operator and only one strategy can be picked. In our case, the emit strategy is decoupled from the aggregation operator itself, and hence the aggregation is computed only once.

Overall, our model covers a wide range in the design space as shown in Figure 5.13. Depending on the emit strategy, any point in the design space as indicated by the blue triangle can be achieved. Our model not only decouples processing latency from result correctness/completeness, but also allows to trade-off processing cost vs. processing latency by applying different emit strategies. In particular, our model allows to use watermark based emit strategies. Therefore, the Dual Streaming Model not only enlarges the design space it can operate in, but also incorporate all trade-offs of watermark based models.

5.3.3 Data Retention

Our model handles out-of-order records by applying a emit-eager-and-update-later strategy. This strategy requires to buffer the history of received input records. For example, stream-stream join operators need to buffer old records to be able to join out-of-order data later. Additionally, our model uses evolving table, which consists of

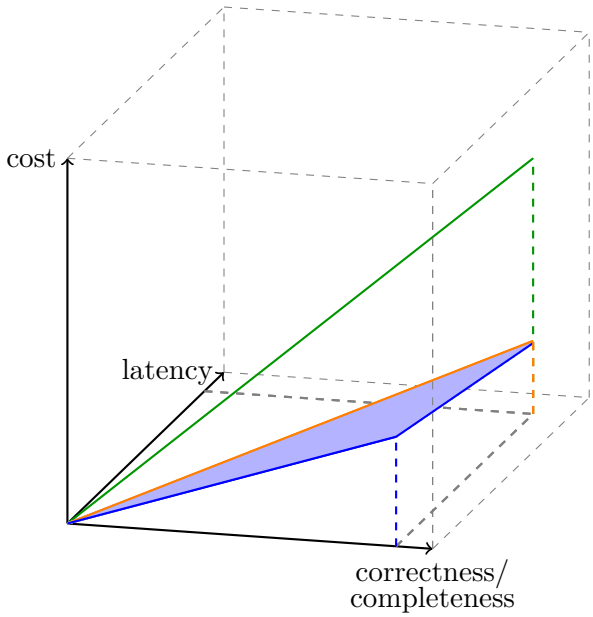

Figure 5.13: Design space of the *Dual Streaming Model*.

table versions. While time progresses new table versions are added. Furthermore, for windowed aggregations, the table primary key space grows without bound over time because new window-IDs are generated while time progresses. Hence, in practice it is required to bound the size of result tables [BGAH07] and internal state.

To avoid unbounded growth of tables and operator state, we apply a so-called *retention period* to all stateful operators and windowed tables. Data that is older than the current time minus the retention period is discarded. Purging data from the result table implies that further updates to those elements are not accepted and corresponding input records are dropped.

Introducing a retention period makes the trade-off between result completeness and storage requirements explicit. For example, if users know that there is an application specific upper bound for the maximum unorder/delay of out-of-order data, they may configure the retention period accordingly to ensure a complete result. An explicit retention period also allow users to reason about memory/storage requirements. As such, retention time is somewhat similar to a punctuation/watermark mechanism [TMSF03, ABB$^+$13], because both are used to purge old state. However, retention period is orthogonal to processing latency and it does not introduce additional latency, because the result table is updated immediately, while a punctuation/watermark would delay the computation until the watermark passes.

5.4 Related Work

In our model, we build on research conducted on sequences, streams, updates, tables, and maintenance of materialized views.

Sequence Databases Sequence processing was studied by Seshadri et al. [SLR94, SLR95, SLR96] introducing the SEQ model. The main abstraction is a sequence of

records with a schema. A sequence is defined as a function ($f : \mathbb{N} \rightarrow \mathbb{S}$) that maps from positions to records, i. e., for each position in the sequence there is exactly one record. Later, this definition is extended to allow for n-to-m mappings between the ordering domain and a set of records. Sequence operators yield result sequences and queries are operator trees. The SEQ model defines selection, projection, position offset (i. e., a "shift" operator), and aggregation operators. Aggregations are define with an `agg_pos`(i) function that returns a set of positions of the input sequence that should be aggregated as well as an aggregation function that is applied to the selected records and computes the result record for the output sequence at position i. Additionally, a *positional* join operator called `compose` is part of the SEQ model.

While ordering is a main concept in SEQ that is used in query optimization, sequences are finite and there is no notion of time. Sequences are stored in the system and it is possible to create indices over sequences. Thus, a sequence can be linearly scanned or probed using an position index. Furthermore, there is no notion of continuous queries, but queries are ad-hoc as in relational database systems. Thus, the SEQ model is closer related to relational database systems than to stream processing, even if the abstraction of ordered records is similar to stream processing.

Continuous Queries in Relational Database Systems Continuous queries over append-only databases were introduced by Terry et al. [TGNO92, GNOT92]. They propose to register queries in a relational database system and execute them periodically. Each time a query is executed, it returns the delta to the result of its previous execution. Even if queries are executed periodically, "the result of a continuous query is the set of data that would be returned if the query were executed at *every* instant in time" (highlighting as in original publication). Terry et al. distinguish between *monotone* and non-monotone queries. Monotone queries guarantee that if a result record is generated, it will be contained in all future results. The paper also introduces rewrite rules to convert a non-monotone query into a monotone one.

The notion of monotone queries is relevant to stream processing. Many stream processing systems only model record streams, and thus can only execute monotone queries correctly. Windowing in stream processing is the most common technique to guarantee monotone queries for input streams with no out-of-order data.

Continual Queries in Relational Database Systems Lui et al. [LPBZ96, LPT99] introduce continual queries that are evaluated periodically in a database system. In contrast to Terry et al. [TGNO92, GNOT92] the database is not restricted to be append-only. The proposed algorithm does not re-evaluate queries from scratch but is able to process the change data. This work is similar to incremental maintenance of materialized view [BLT86].

Incremental updates are a core concept in our model. The work of Lui et al. and the view maintenance problem in relational databases are defined on relations (i. e., finite input data) and thus are not directly applicable to data streams. However, it is closely related to our model with regard to evolving tables.

Chronicle Data Model The chronicle data model [JMS95] builds on chronicles, relations, and persistent views. A chronicle is an unbounded sequence of transaction

records and only the tail of the sequence containing the latest records is stored. Transactional records with the same sequence number are grouped in a chronon that has a version number that is equal to the record sequence number. Relations are similarly defined as in the relational model with the addition of version numbers. Persistent views are relations without a version number and the result of operators that remove the sequence attribute. The chronicle algebra is defined on version numbers (i. e., join between a chronicle and a relation) and includes operators similar to the relational model (e. g., selection, projection, join, group-by, and aggregation). Since only the tail of chronicles is stored, the chronicle algebra and its operators are defined with monotonic semantics with respect to insertions into the chronicles. An extension of the chronicle data model adds a time domain that allows to define time based views that are similar to time windows in stream processing.

The chronicle data model is quite similar to the model we propose in this work. However, the time domain is only part of the extended model and not a first class citizen as in our model. Furthermore, our model includes out-of-order record processing and contains the notion of a changelog stream, which enables defining non-monotonic operators.

Updates in Data Streams Data streams containing update or delete records have been considered in previous work. Babu and Widom [BW01] discuss the general idea of updates and deletes in data streams. However, they do not provide detailed operator semantics or define the correctness of results formally.

The Borealis system [AAB+05, RMCZ06] applies a "time travel" approach to process updates by storing the history of the original input stream. If an update record occurs, the stored input stream is updated accordingly (i. e., an in-place update) and reprocessed using the new record. To reduce downstream updates, it does not re-emit all newly computed output messages, but only the deltas to the previous output, using the stored history of the output stream. Additionally, it allows for downstream result refinement via revision tuples that send old and new values. Compared to our model, Borealis is still a stream-only system and does not support tables. Also, refinements are not defined formally. It is unclear how long a stream history is preserved, or how refinement tuples are mapped to existing tuples. Furthermore, reprocessing input streams suffixes after updating them is compute intensive compared to our incremental updating approach.

Semantic Models The CQL continuous query language builds on the relational model to provide strong semantics. Data streams are always converted into tables using special operators and actual processing is done via the relational model using those tables. CQL also introduces operators to convert tables back to data streams. Records in the CQL model have a relational schema and a timestamp, which is not part of the schema. Windowing is supported by corresponding stream-to-relation operators.

Our model as well as CQL support streams and tables as first class citizens. The difference is that processing is always based on tables in CQL, even for stateless transformations like map, projection, or filtering/selection. Those transformations return a data stream in our model but a table in CQL that must be converted into a stream by the user. Some stream-table transformations are done by the system auto-

matically, making it even harder for users to understand the semantics of the system because those transformation rules are quite complex. Transforming streams into tables for stateless transformations also has the disadvantage that records might be lost if they have the same timestamp in a non-deterministic manner (c. f. [JMS+08]). Another difference is the types of supported streams: CQL used `IStream` for insert streams, `DStream` for delete stream, and `RStream` that is the relation stream. In our model, we support record streams (similar as `IStream`) and changelog streams (a combination of `IStream` and `Dstream`). It is important to note that a `RStream` can be used to compute a corresponding changelog stream and thus both models are equally expressive with regard to their types of data streams.

While the CQL model was the first to define strict operator semantics, the model seems to be quite attached to the relational model. In contrast to CQL, we propose to embrace data streams as first class citizens in combination with relational tables. The goal is to simplify operator semantics and allow users to reason about the system easily.

Law et al. [LWZ04] introduce a similar model to ours and define operators that continuously update the output with the notion of "the result so far". They define correctness of operators based on input prefix and have a formal notion of blocking and non-blocking operators. The difference to our work is that they only model record streams and their model is limited to monotonic queries for this reason. They also do not consider out-of-order records or windowing operators.

The SECRET model [DTM+13] aims to describe different window semantics with a uniform model. SECRET focuses on centralized stream processing systems, cannot express out-of-order data, and does not cover other stream processing operators [ATM+17]. Our model is more generic than SECRET, which focuses solely on windowing semantics.

Order and Time Order and time present multiple challenges in data stream processing. For example, how to handle out-of-order data and what time semantics should be used? The notion of event-time vs. processing-time was first introduced by Srivastava and Widom [SW04]. They noted that processing-time or ingestion-time guarantees that there is no out-of-order data. Using event-time raises challenges for unsynchronized clocks of external data sources resulting in skewed time, delays, and out-of-order data. Buffering and reordering is one suggested technique to reorder data. For avoiding delay, heartbeats are introduced, which also help to detect unsynchronized clocks.

Time semantics are discussed by Barga et al. [BGAH07]. They introduce strong time semantics similar to temporal database systems, including definitions for different levels of consistency. Out-of-order records as well as blocking operators are considered, too. A temporal-relational algebra is also used by StreamScope [LFQ+16] based on time intervals that are assigned to records instead of scalar timestamps.

Another approach to handle out-of-order data are *punctuations* [TMSF03]. Punctuations are control messages that provide certain guarantees about the data stream. For example, they can express that no record after the punctuation will have a timestamp smaller than a certain value. Thus, punctuations are similar to heartbeats but more generic as they can express arbitrary constraints on the data, while heartbeats only express time progress.

The discussed techniques have in common that they imply a partially blocking computation until a heartbeat or punctuation arrives, or a buffer to reorder records is filled up. Thus, those techniques result in delays and increased processing latency. A different approach is to process all data immediately to keep processing latency as small as possible and refine results later if required. We follow the update approach and use punctuations to trade-off space (reduced number of downstream updates) vs. time (increase latency). Thus, updates are a more generic approach compared to punctuations and partial blocking.

Finally, Bogeli et al. [BAH+19] propose a change to SQL that allows to incorporate data streams and temporal tables and to express unified queries over both. Their temporal table is very similar to our evolving table, however, their processing model is not based on continuous updates as ours, but on watermarks and triggers.

5.5 Summary

We introduced the *Dual Streaming Model* in the second part of this thesis. It puts forward the duality between data streams and temporal-relational tables, and treats state as first class citizen instead of an internal implementation detail. To this end, the Dual Streaming Model decouples result correctness/completeness from processing latency. This decoupling opens up the design space for data stream processing applications and allow users to trade-off result correctness/completeness vs. processing latency vs. processing cost. Furthermore, our model enables users to retrieve the result of their program either push based, by subscribing to the result stream, or pull based by querying the result table.

The Dual Streaming Model is the foundation of *Kafka Streams* [ASFg], the stream processing library of Apache Kafka [ASFc, KNR11]. Kafka Streams is widely adopted in the industry, including large enterprises, which shows that our Dual Streaming Model is useful in practice.

Part IV

Discussion

Chapter 6

Conclusion

The requirement for low-latency data processing of high-volume data streams had increased over the last few years. Yet, state-of-the-art distributed stream processing systems are still hard to operate in practice. Furthermore, there is no agreement on a unified processing model, and different systems offer different semantics and trade-offs to the user.

In this thesis, we introduced a cost-model for data-parallel distributed stream processing systems (Chapter 3). Our cost-model is built on operator parallelism and record batching. To execute a streaming data flow program efficiently, record batching may be employed to trade-off processing latency vs. throughput. Furthermore, for high-volume data streams, data-parallelism is used to allow a system to process all data without "falling behind". Our model considers CPU and network cost to estimate the required degree of parallelism for each operator given a target input data rate. Based on our cost-model, we presented multiple algorithms to detect processing bottlenecks, predict the data flow throughput, and to optimize batch sizes as well as operator parallelism (Chapter 4). To this end, we believe that our cost-model and analytical optimization approach should be combined with dynamic scaling to adapt to the changing characteristics of input data streams. Furthermore, extending our model to incorporate processing latency and to extend dynamic batching approaches with such a model is interesting future work.

In the second part of this thesis (Chapter 5), we proposed the *Dual Streaming Model* that unifies concepts of existing models. We put forward the duality of data streams and temporal-relational tables, and treat state as a first class citizen. The model makes explicit to the user, the trade-off between processing cost vs. processing latency vs. result correctness/completeness. Hence, it opens the design space for stream processing applications and allows users to pick a trade-off depending on their application requirements. We believe that the Dual Streaming Model is a step forward to generic processing semantics that allow to express a wide variety of stream processing application within a single model.

Bibliography

[AA91] N. Wilschut A. and Peter M. G. Apers. *Dataflow Query Execution in a Parallel Main-Memory Environment*, pages 68–77. IEEE Computer Society, United States, 12 1991. Imported from EWI/DB PMS [db-utwente:arti:0000002032].

[AAB+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the Borealis stream processing engine. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 277–289, 2005.

[AAB+06] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. SPC: A distributed, scalable platform for data mining. In *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms*, DMSSP '06, pages 27–37, New York, NY, USA, 2006. ACM.

[ABB+03a] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. STREAM: The Stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.

[ABB+03b] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The Stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665, New York, NY, USA, 2003. ACM.

[ABB+13] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, August 2013.

[ABC+05] Yanif Ahmad, Bradley Berg, Uğur Cetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alexander Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stan Zdonik. Distributed operation in the Borealis stream processing engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 882–884, New York, NY, USA, 2005. ACM.

[ABC+15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceeding of the VLDB Endowment*, 8(12):1792–1803, August 2015.

[ABQ13] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in Storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 207–218, New York, NY, USA, 2013. ACM.

[ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A language for continuous queries over streams and relations. In *Database Programming Languages, 9th International Workshop, DBPL 2003, Potsdam, Germany, September 6-8, 2003, Revised Papers*, pages 1–19, 2003.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[ACc+03a] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: A data stream management system. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 666–666, New York, NY, USA, 2003. ACM.

[ACc+03b] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003.

[ACG+04] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 480–491. VLDB Endowment, 2004.

[ADT+18] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative API for real-time applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 601–613, New York, NY, USA, 2018. ACM.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. *SIGMOD Records*, 29(2):261–272, May 2000.

[AJS+06] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive control of extreme-scale stream processing systems. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006), 4-7 July 2006, Lisboa, Portugal*, page 71, 2006.

[AN04] Ahmed M. Ayad and Jeffrey F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 419–430, New York, NY, USA, 2004. ACM.

[ASFa] The Apache Software Foundation. Apache Flink project web page. `https://flink.apache.org/`.

[ASFb] The Apache Software Foundation. Apache Heron project web page. `https://apache.github.io/incubator-heron/`.

[ASFc] The Apache Software Foundation. Apache Kafka project web page. `https://kafka.apache.org/`.

[ASFd] The Apache Software Foundation. Apache S4 project web page. `https://incubator.apache.org/projects/s4.html`.

[ASFe] The Apache Software Foundation. Apache Samze project web page. `https://samza.apache.org/`.

[ASFf] The Apache Software Foundation. Apache Storm project web page. `https://storm.apache.org/`.

[ASFg] The Apache Software Foundation. Kafka Streams documenation. `https://kafka.apache.org/documentation/streams/`.

[ATM+17] Lorenzo Affetti, Riccardo Tommasini, Alessandro Margara, Gianpaolo Cugola, and Emanuele Della Valle. Defining the execution semantics of stream processing engines. *Journal of Big Data*, 4(1):12, Apr 2017.

[AW04] Arvind Arasu and Jennifer Widom. A denotational semantics for continuous queries over streams and relations. *SIGMOD Records*, 33(3):6–11, September 2004.

[BAH+19] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. One SQL to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1757–1772. ACM, 2019.

[BBC+04] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stan Zdonik. Retrospective on Aurora. *The VLDB Journal*, 13(4):370–383, December 2004.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[BBD+04] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):333–353, December 2004.

[BBMD03] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. Chain: Operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 253–264, New York, NY, USA, 2003. ACM.

[BBMS05] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 13–24, New York, NY, USA, 2005. ACM.

[BBMS08] Magdalena Balazinska, Hari Balakrishnan, Samuel R. Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Transactions on Database Systems*, 33(1):3:1–3:44, March 2008.

[BBS04] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Load management and high availability in the Medusa distributed stream processing system. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 929–930, New York, NY, USA, 2004. ACM.

[BCG+11] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-

intensive computing. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1151–1162, Washington, DC, USA, 2011. IEEE Computer Society.

[BDD+10] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J. Miller, and Nesime Tatbul. SECRET: A model for analysis of the execution semantics of stream processing systems. *Proceedings of the VLDB Endowment*, 3(1-2):232–243, September 2010.

[BEH+10] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 119–130, New York, NY, USA, 2010. ACM.

[BFc12] Nathan Backman, Rodrigo Fonseca, and Uğur Çetintemel. Managing parallelism for stream processing in the cloud. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, HotCDP '12, pages 1:1–1:5, New York, NY, USA, 2012. ACM.

[BGAH07] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 363–374, 2007.

[BH02] Richard J. Bolton and David J. Hand. Statistical fraud detection: A review. *Statistical Science*, 17(3):235–249, 2002.

[BHL+10] Dominic Battré, Matthias Hovestadt, Björn Lohrmann, Alexander Stanik, and Daniel Warneke. Detecting bottlenecks in parallel dag-based data flow programs. In *3rd Workshop on Many-Task Computing on Grids and Supercomputers, MTAGS@SC 2010, New Orleans, Louisiana, USA, November 15, 2010*, pages 1–10. IEEE Computer Society, 2010.

[BLT86] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. *SIGMOD Records*, 15(2):61–71, June 1986.

[BROL14] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. Summingbird: A framework for integrating batch and online MapReduce computations. *Proceedings of the VLDB Endowment*, 7(13):1441–1451, August 2014.

[BW01] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Records*, 30(3):109–120, September 2001.

[BW04] Shivnath Babu and Jennifer Widom. StreaMon: An adaptive engine for stream query processing. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 931–932, New York, NY, USA, 2004. ACM.

[CBB+03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Uğur Çetintemel, Ying Xing, and Stanley B. Zdonik. Scalable distributed stream processing. In *CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*, 2003.

[CCA+10] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[CcC+02] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: A new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 215–226. VLDB Endowment, 2002.

[CCD+03a] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*, 2003.

[CCD+03b] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[CcR+03] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 838–849. VLDB Endowment, 2003.

[CDE+16] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Peng, and Paul Poulosky. Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*, pages 1789–1792. IEEE Computer Society, 2016.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for internet databases.

In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 379–390, New York, NY, USA, 2000. ACM.

[CEF+17] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in Apache Flink®: Consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, August 2017.

[CER17] CERN. Future ICT challenges in scientific research. `http://cds.cern.ch/record/2301895/files/Whitepaper_brochure_ONLINE.pdf`, 2017.

[CF02] Sirish Chandrasekaran and Michael J. Franklin. Streaming queries over streaming data. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 203–214. VLDB Endowment, 2002.

[CFMKP13] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.

[CGB+14] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proceeding of the VLDB Endowment*, 8(4):401–412, December 2014.

[CGJ+02] Chuck Cranor, Yuan Gao, Theodore Johnson, Vlaidslav Shkapenyuk, and Oliver Spatscheck. Gigascope: High performance network monitoring with an SQL interface. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 623–623, New York, NY, USA, 2002. ACM.

[CJK+04] Graham Cormode, Theodore Johnson, Flip Korn, S. Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. Holistic UDAFs at streaming speeds. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 35–46, New York, NY, USA, 2004. ACM.

[CJSS03a] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. The Gigascope stream database. *IEEE Data Engineering Bulletin*, 26(1):27–32, 2003.

[CJSS03b] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 647–651, New York, NY, USA, 2003. ACM.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 38(4):28–38, 2015.

[CKSV08] Michael Cammert, Jürgen Krämer, Bernhard Seeger, and Sonny Vaupel. A cost-based approach to adaptive resource management in data stream systems. *IEEE Transactions on Knowledge Data Engineering*, 20(2):230–245, 2008.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, February 1985.

[Cla15] Peter Clay. A modern threat response framework. *Network Security*, 2015(4):5–10, 2015.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[CRP+10] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. *SIGPLAN Notices*, 45(6):363–375, June 2010.

[CWI+16] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Realtime data processing at facebook. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1087–1098, New York, NY, USA, 2016. ACM.

[Dea06] Jeffrey Dean. Experiences with MapReduce, an abstraction for large-scale computation. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 1–1, New York, NY, USA, 2006. ACM.

[Des04] Amol Deshpande. An initial study of overheads of eddies. *SIGMOD Records*, 33(1):44–49, March 2004.

[DG92] David DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[DGGR02] Alin Dobra, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 61–72, New York, NY, USA, 2002. ACM.

[DLB+11] Michael Daum, Frank Lauterwald, Philipp Baumgärtel, Niko Pollner, and Klaus Meyer-Wegener. Black-box determination of cost models' parameters for federated stream-processing systems. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, IDEAS '11, pages 226–232, New York, NY, USA, 2011. ACM.

[DRV03] Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors. *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India.* IEEE Computer Society, 2003.

[DTM+13] Nihal Dindar, Nesime Tatbul, Renée J. Miller, Laura M. Haas, and Irina Botan. Modeling the execution semantics of stream processing engines with SECRET. *The VLDB Journal*, 22(4):421–446, August 2013.

[DZSS14] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 16:1–16:13, New York, NY, USA, 2014. ACM.

[FAG+17] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: Self-regulating stream processing in Heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, August 2017.

[FDM+15] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. DRS: Dynamic resource scheduling for real-time analytics over fast streams. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, June 29 - July 2, 2015*, pages 411–420, 2015.

[FDM+17] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. DRS: Auto-scaling for real-time stream analytics. *IEEE/ACM Transactions on Networking*, 25(6):3338–3352, December 2017.

[GAW+08] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. SPADE: The System S declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD International*

Conference on Management of Data, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[GC06] Joseph S. Gomes and Hyeong-Ah Choi. Cost-based solution for optimizing multi-join queries over distributed streaming sensor data. In Enrico Blanzieri and Tao Zhang, editors, *2nd International ICST Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom 2006, Atlanta, GA, USA, November 17-20, 2006*. IEEE Computer Society / ICST, 2006.

[GCB+97] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, January 1997.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[GJPPM+12] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. StreamCloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, December 2012.

[GJPPMV10] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, and Patrick Valduriez. StreamCloud: A large scale data streaming system. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, pages 126–137, Washington, DC, USA, 2010. IEEE Computer Society.

[GNOT92] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, December 1992.

[GO03a] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Records*, 32(2):5–14, June 2003.

[GO03b] Lukasz Golab and M Tamer Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 500–511. VLDB Endowment, 2003.

[Gul12] Vincenzo Gulisano. *StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine. (StreamCloud: un moteur de traitement de streams parallèle et distribué)*. PhD thesis, Technical University of Madrid, Spain, 2012.

[HAE08] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. Query processing of multi-way stream window joins. *The VLDB Journal*, 17(3):469–488, May 2008.

[HCCZ08] Jeong-Hyon Hwang, Sanghoon Cha, Uğur Cetintemel, and Stan Zdonik. Borealis-R: A replication-transparent stream processing system for wide-area monitoring applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1303–1306, New York, NY, USA, 2008. ACM.

[HJHF14] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 13–22, New York, NY, USA, 2014. ACM.

[HSS+14] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys*, 46(4):46:1–46:34, March 2014.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[JAA+06] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the Stream Processing Core. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 431–442, New York, NY, USA, 2006. ACM.

[JMR05] Theodore Johnson, S. Muthukrishnan, and Irina Rozenbaum. Sampling algorithms in a stream operator. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 1–12, New York, NY, USA, 2005. ACM.

[JMS95] H. V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz. View maintenance issues for the chronicle data model (extended abstract). In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '95, pages 113–124, New York, NY, USA, 1995. ACM.

[JMS+08] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment*, 1(2):1379–1390, August 2008.

[JMSS05] Theodore Johnson, S. Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. A heartbeat mechanism and its application in Gigascope. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1079–1088. VLDB Endowment, 2005.

[KBF+15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[KCC+03] Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Samuel Madden, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: An architectural status report. *IEEE Data Engineering Bulletin*, 26(1):11–18, 2003.

[KFD+10] Sailesh Krishnamurthy, Michael J. Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. Continuous analytics over discontinuous streams. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1081–1092, New York, NY, USA, 2010. ACM.

[KK15] Martin Kleppmann and Jay Kreps. Kafka, Samza and the unix philosophy of distributed data. *IEEE Data Engineering Bulletin*, 38(4):4–14, 2015.

[KNR11] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[KNV03] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In Dayal et al. [DRV03], pages 341–352.

[KRK+18] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1507–1518. IEEE Computer Society, 2018.

[KS09] Jürgen Krämer and Bernhard Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Transactions on Database Systems*, 34(1):4:1–4:49, April 2009.

[LFQ+16] Wei Lin, Haochuan Fan, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. StreamScope: Continuous reliable distributed processing of big data streams. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 439–453, 2016.

[LJK15] Björn Lohrmann, Peter Janacik, and Odej Kao. Elastic stream processing with latency guarantees. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus,*

OH, USA, June 29 - July 2, 2015, pages 399–410. IEEE Computer Society, 2015.

[LLP+12] Wang Lam, Lu Liu, Sts Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. Muppet: MapReduce-style processing of fast data. *Proceedings of the VLDB Endowment*, 5(12):1814–1825, August 2012.

[LMT+05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 311–322, 2005.

[LPBZ96] Ling Liu, Calton Pu, Roger S. Barga, and Tong Zhou. Differential evaluation of continual queries. In *Proceedings of the 16th International Conference on Distributed Computing Systems, Hong Kong, May 27-30, 1996*, pages 458–465, 1996.

[LPT99] Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge Data Engineering*, 11(4):610–628, 1999.

[LTS+08] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: A new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment*, 1(1):274–288, August 2008.

[LWK12] Björn Lohrmann, Daniel Warneke, and Odej Kao. Massively-parallel stream processing under QoS constraints with Nephele. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 271–282, New York, NY, USA, 2012. ACM.

[LWK14] Björn Lohrmann, Daniel Warneke, and Odej Kao. Nephele streaming: stream processing under QoS constraints at scale. *Cluster Computing*, 17(1):61–78, 2014.

[LWZ04] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Query languages and data models for database sequences and data streams. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 492–503. VLDB Endowment, 2004.

[MF02] Samuel Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 555–566, 2002.

[MLT+05] David Maier, Jin Li, Peter Tucker, Kristin Tufte, and Vassilis Papadimos. Semantics of data streams and operators. In *Proceedings of the 10th International Conference on Database Theory*, ICDT'05, pages 37–52, Berlin, Heidelberg, 2005. Springer-Verlag.

[MMI+13] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[MSHR02] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 49–60, New York, NY, USA, 2002. ACM.

[MWA+03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*, 2003.

[NMG+15] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David García-Soriano, Nicolas Kourtellis, and Marco Serafini. The power of both choices: Practical load balancing for distributed stream processing engines. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 137–148. IEEE Computer Society, 2015.

[NPP+17] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, August 2017.

[NRNK10] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177, Washington, DC, USA, 2010. IEEE Computer Society.

[OV99] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[PHH+15] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. R-Storm: Resource-aware scheduling in Storm. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 149–161, New York, NY, USA, 2015. ACM.

[RDH03] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In Dayal et al. [DRV03], pages 353–364.

[RMCZ06] Esther Ryvkina, Anurag Maskey, Mitch Cherniack, and Stanley B. Zdonik. Revision processing in a stream processing engine: A high-level design. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 141. IEEE Computer Society, 2006.

[RSW+07] Frederick Reiss, Kurt Stockinger, Kesheng Wu, Arie Shoshani, and Joseph M. Hellerstein. Enabling real-time querying of live and historical stream data. In *19th International Conference on Scientific and Statistical Database Management, SSDBM 2007, 9-11 July 2007, Banff, Canada, Proceedings*, page 28. IEEE Computer Society, 2007.

[ScZ05] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Records*, 34(4):42–47, December 2005.

[SGH15] Scott Schneider, Bugra Gedik, and Martin Hirzel. Language runtime and optimizations in IBM streams. *IEEE Data Engineering Bulletin*, 38(4):61–72, 2015.

[SH98] Mark Sullivan and Andrew Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '98, pages 2–2, Berkeley, CA, USA, 1998. USENIX Association.

[SHCF03] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In Dayal et al. [DRV03], pages 25–36.

[SLR94] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 430–441, New York, NY, USA, 1994. ACM.

[SLR95] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. SEQ: A model for sequence databases. In *Proceedings of the Eleventh International Conference on Data Engineering*, ICDE '95, pages 232–239, Washington, DC, USA, 1995. IEEE Computer Society.

[SLR96] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The design and implementation of a sequence database system. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 99–110, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[STD+00] Jayavel Shanmugasundaram, Kristin Tufte, David J. DeWitt, David Maier, and Jeffrey F. Naughton. Architecting a network query engine for producing partial results. In *The World Wide Web and Databases, Third International Workshop WebDB 2000, Dallas, Texas, USA, Maaay 18-19, 2000, Selected Papers*, pages 58–77, 2000.

[Sul96] Mark Sullivan. Tribeca: A stream database manager for network traffic analysis. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 594–, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[SW04] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '04, pages 263–274, New York, NY, USA, 2004. ACM.

[SY93] James W. Stamos and Honesty C. Young. A symmetric fragment and replicate algorithm for distributed joins. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1345–1354, 1993.

[TGNO92] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, pages 321–330, New York, NY, USA, 1992. ACM.

[TMSF03] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, March 2003.

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[VN02] Stratis D. Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 37–48, New York, NY, USA, 2002. ACM.

[VPO+17] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 374–389, New York, NY, USA, 2017. ACM.

[WBH+08] Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. SODA: An

optimizing scheduler for large-scale stream-based distributed computer systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 306–325, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[WKWO12] Sai Wu, Vibhore Kumar, Kun-Lung Wu, and Beng Chin Ooi. Parallelizing stateful operators in a distributed stream processing system: How, should you and how much? In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 278–289, New York, NY, USA, 2012. ACM.

[XPG16] Le Xu, Boyang Peng, and Indranil Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *2016 IEEE International Conference on Cloud Engineering, IC2E 2016, Berlin, Germany, April 4-8, 2016*, pages 22–31. IEEE Computer Society, 2016.

[XY07] Junyi Xie and Jun Yang. A survey of join processing in data streams. In Charu C. Aggarwal, editor, *Data Streams - Models and Algorithms*, volume 31 of *Advances in Database Systems*, pages 209–236. Springer, 2007.

[XZH05] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the Borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 791–802, Washington, DC, USA, 2005. IEEE Computer Society.

[YDHP07] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: Simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, 2007. ACM.

[YM15] Mansheng Yang and Richard T.B. Ma. Smooth task migration in Apache Storm. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 2067–2068, New York, NY, USA, 2015. ACM.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[ZDL+13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[ZSC+03] Stanley B. Zdonik, Michael Stonebraker, Mitch Cherniack, Uğur Çetintemel, Magdalena Balazinska, and Hari Balakrishnan. The Aurora and Medusa projects. *IEEE Data Engineering Bulletin*, 26(1):3–10, 2003.